The hardware implementation of an artificial neural network using stochastic pulse rate encoding principles

Glover, John Sigsworth

The Hardware Implementation Of An Artificial Neural Network Using Stochastic Pulse Rate Encoding Principles

John Sigsworth Glover
M.Eng. (Leeds)

School of Engineering
University of Durham

A thesis submitted in partial fulfillment of the requirements of the Council of the University of Durham for the Degree of Doctor of Philosophy (Ph.D.).

September 1995

Abstract

In this thesis the development of a hardware artificial neuron device and artificial neural network using stochastic pulse rate encoding principles is considered.

After a review of neural network architectures and algorithmic approaches suitable for hardware implementation, a critical review of hardware techniques which have been considered in analogue and digital systems is presented. New results are presented demonstrating the potential of two learning schemes which adapt by the use of a single reinforcement signal.

The techniques for computation using stochastic pulse rate encoding are presented and extended with new novel circuits relevant to the hardware implementation of an artificial neural network. The generation of random numbers is the key to the encoding of data into the stochastic pulse rate domain. The formation of random numbers and multiple random bit sequences from a single PRBS generator have been investigated. Two techniques, Simulated Annealing and Genetic Algorithms, have been applied successfully to the problem of optimising the configuration of a PRBS random number generator for the formation of multiple random bit sequences and hence random numbers.

A complete hardware design for an artificial neuron using stochastic pulse rate encoded signals has been described, designed, simulated, fabricated and tested before configuration of the device into a network to perform simple test problems. The implementation has shown that the processing elements of the artificial neuron are small and simple, but that there can be a significant overhead for the encoding of information into the stochastic pulse rate domain. The stochastic artificial neuron has the capability of on-line weight adaption. The implementation of reinforcement schemes using the stochastic neuron as a basic element are discussed.

Acknowledgements

The following people have been vital to the production of this thesis, but many others have also contributed to my welfare during the development of this thesis.

- Professor Phil Mars of the University of Durham – for all his guidance and advice.
- Dr Simon Johnson of the University of Durham – for discussions on the hardware implementation techniques available at the University of Durham.
- University of Teesside – for their assistance in the fabrication of the artificial neuron devices.
- Raghu, Chen, Alan, Jeremy, David, Matthew, Martin and Stephen, my colleagues in the lab for their support.

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree, and that all sources of information have been duly acknowledged.

Contents

List of Figures

List of Abbreviations

ACE	Adaptive Critic Element
ADDIE	Adaptive Digital Element
AE	Adaptive Element
ANN	Artificial Neural Network
ASE	Associative Search Element
ASIC	Application Specific Integrated Circuit
ASN	Associative Search Network
BAM	Bidirectional Associative Memory
CAM	Content-Addressable Memory
CDF	Cumulative Distribution Function
DLB	Dual Line Bipolar
GA	Genetic Algorithm
GUI	Graphical User Interface
HDL	Hardware Description Language
IC	Integrated Circuit
LMS	Least Mean Square
MLP	Multi Layer Perceptron
NN	Neural Network
PDF	Probability Density Function
PE	Predictor Element
PRBS	Pseudo Random Binary Sequence
SA	Simulated Annealing
SLB	Single Line Bipolar
SLU	Single Line Unipolar
TDNN	Time-delay Neural Network
URN	Uniform Random Number
VFSR	Very Fast Simulated Re-annealing
VLSI	Very Large Scale Integration

Chapter 1

Introduction

The art of computing is, as ever, advancing rapidly with new architectures for machines and processors, new fabrication techniques for components which enable a reduction in size and an increase in the speed of operation occurring all the time. Programming languages and operating systems are becoming more tractable and user friendly, command line user interfaces are being superceded by graphical user interfaces. However, these machines still adopt a *conventional* approach, based upon a von Neumann architecture, of an inherently complex central processing unit and attached memory. There are parallel processing systems available which may have several processing units operating concurrently either on shared or individual memory but these systems must still be explicitly programmed to operate. Despite these advances in speed and sophistication certain tasks still remain difficult to program a machine to perform effectively, eg. speech, vision, reasoning or contents based information processing tasks. However these are tasks which are performed regularly and with ease by animals.

The structure of the information processing system in animals is different. The brain and nervous system which performs these tasks is based upon what is thought to be a basic processing unit, the neuron, in a massively parallel architecture, with a high level of interconnectivity, distributed memory and a relatively slow speed of operation. In addition this system is not explicitly programmed to perform but can learn and adapt to new situations, experiences and environments.

The reliability and fault tolerance of the two different approaches is interesting to note. For traditional systems a component or sub-system failure is usually catastrophic until repaired leading to multiple systems being operated in parallel for safety critical tasks. Networks of neurons are generally fault tolerant with their large number of processing elements and interconnections. In fact, the system is constantly evolving as it operates with cells dying and new ones being added.

There therefore must be merit in this alternative method of approach to information processing and thus there is a desire to study, simulate and model these approaches which do not need to be programmed to perform a task but can be trained and which have the potential to be fault tolerant. The study of networks of neurons is widespread and conducted in many different fields across science and engineering including electronics, computing, optics, biology and psychology. The generic title to this area is usually Neural Networks and in the particular case of synthetic systems Artificial Neural Networks.

The study of neural networks could be approached in several ways: the investigation of learning algorithms, the study of the biochemistry of living neural networks, the examination of decision making systems or the development of simplified plausible models in software and hardware. From an engineering point of view not all of these are relevant approaches. The study of software and hardware neural network models and implementation is pertinent to engineering since ultimately any feasible system must be developed and operated.

Much work has been conducted into learning and adaption algorithms with systems which will adapt their behaviour based upon either the system's own experience or by external influence from the environment. Often incorporated into these systems is a model of a neuron usually based upon the principle of a function of a weighted sum of inputs. The system is often simulated in software upon a conventional machine for the relative ease that this offers in varying the system and model. For development and research purposes this is often adequate. If, however, an operational system is required with a practical real time response the issue of fabricating such developed algorithms and networks in hardware must be considered which is what this thesis sets out to address.

In realising a hardware artifical neural network system several issues must be addressed:

The algorithm and neural network system architecture to be adopted.

Many architectures and algorithms have been, and are continuing to be proposed. However several approaches, particularly the more sophisticated, are not necessarily suitable for the development of a dedicated hardware solution for individual processing elements. In addition, the learning and adaption algorithm may not be easily integrated into a hardware environment. This does not mean that these systems are without merit but that they are not currently appropriate for the development of hardware.

The system to be used for building the network.

System realisation could be undertaken in many different fields, eg. electronics, optical or perhaps even biological. The latter two fields may be interesting but are not pertinent for this work, for the electronics approach the assorted analogue and digital methods should be assessed.

The signalling and communication methods to be adopted.

The method of signalling and control is allied strongly to the approach adopted for the main hardware realisation.

The provision for on-line learning, adaption or adjustment of performance.

If a neural network is constructed in hardware is its performance determined at build time, run time or can it be adapted as it operates? Ideally the latter method should be feasible but probably bootstrapping the system by the programming of a base configuration in the network should be enabled.

The effectiveness with which the architecture can be extended or reconfigured in the selected hardware.

Is the hardware easy to reconnect into a new configuration? Can the inputs to hardware devices be adjusted for different architectures and could the number of neurons in the system be varied easily still allowing the system to trained and operate effectively.

How the approach taken could be enhanced.

Finally, is the hardware implementation the only one feasible or is it possible to enhance the system to improve the performance or correct mistakes, ie. does the basic approach work. This can only really be answered by constructing and demonstrating the capabilities of a system.

The above issues will be addressed as outlined in the following section, with the selected hardware solution of stochastic pulse rate encoding explained, justified and implemented.

1.1 Outline of Thesis

In this thesis issues relating to the hardware implementation of an artifical neuron and an artificial neural network using stochastic pulse rate encoding principles are discussed. The aim is to present a potential solution to the problem of realising artificial neurons in hardware since most work is currently conducted via software synthesis and modelling. The outline of the thesis structure is consequently presented below.

In Chapter 2 a review of ANN architectures and algorithms which display a relevance to hardware implementation is presented. Validation for two of these systems is conducted, the Multi-layer Perceptron and the Kohonen Self-Organising Feature Map, and the scheme of reinforcement learning using A_{R-P} techniques is extended to form two new models which just use a single reinforcement feedback connection for adaption purposes. Chapter 3 provides a critical review of hardware implementation systems and describes some currently available dedicated hardware devices. Within this chapter pulse rate encoding strategies are introduced, but with a full discussion of stochastic pulse rate encoding techniques deferred to Chapter 4. Included in the critical review of Chapter 4 into stochastic pulse rate encoded processing is the presentation of new novel circuits with relevance to the implementation of a neuron using these techniques. Chapter 5 discusses issues relating to formation of multiple random number sequences from a single PRBS generator. The two optimisation techniques of Simulated Annealing and Genetic Algorithms are presented and applied to the problem of the optimum configuration determination for the PRBS and its ancillary circuitry. Chapter 6 draws together the techniques and issues raised in the preceding chapters to enable the design of an artificial neuron operating upon stochastic pulse rate encoded signals to be presented. The neuron design is described and has been fabricated enabling the testing and subsequent analysis of its operation in a limited network to be described. This thesis is concluded in Chapter 7 with a summary of the results presented and suggestions for further work.

Chapter 2

Aspects of Artificial Neural Networks

In this chapter a critical review is provided of the some of the key types of neural networks which have been developed together with associated training algorithms and strategies. The following types of network are explained with the aim of gaining an understanding of different approaches taken in this field and to determine the most appropriate system for hardware implementation with an on-line learning algorithm.

Perceptron, MLP and Backpropagation. This type of network is one of the most widely used and provides feedforward connections only through the network. A feedforward network will ultimately be demonstrated using the designed hardware neuron of §6.

Kohonen Self-Organising Feature Map. This network was investigated since it does not require external intervention in the learning process but is able to adapt itself to the task it will perform.

Hopfield Net. This network introduces the concept of feedback connections and highlights the property that energy minimisation within a neural network architecture is relevant to the learning process.

Boltzmann Machine. Learning and adaption through random processes are demonstrated to be achievable and valuable by the study of the Boltzmann machine. The hardware neuron developed later will use a stochastic signalling strategy to perform inter-neuron communication and computation.

Reinforcement Learning and $\mathbf{A_{R-P}}$. Simple learning strategies in which only a single signal is fed back to the processing elements are reviewed. The A_{R-P} strategies, in particular, are relevant since they provide the basis for algorithms which may be combined with the hardware neuron developed to produce an integrated performance.

The area of reinforcement learning is expanded upon in this chapter. After an initial validation of the work of Barto *et al*, [2], into A_{R-P}, two extensions to the learning strategies called the Q-model and T-model A_{R-P} are proposed and tested. Results are presented demonstrating the ability of these new algorithms to adapt and solve basic feedforward problems.

2.1 The Biological Inspiration for Artificial Neural Networks.

Artificial neural systems, neurocomputers, connectionist models, parallel distributed processing models, layered self-adaptive systems, self-organising systems, neuromorphic systems and cyberware are all terms which can be applied to a technology and ideology which can be encompassed under the title of Artificial Neural Networks (ANN) or just Neural Networks (NN). The roots and inspiration for ANNs are drawn from biology and biological nervous systems. Such biological systems or *wetware* consists of a multitude of simple processing elements which are connected together in a massively parallel architecture.

The brain consists of many neurons of different varieties but following the general format as illustrated in Figure 2.1. A formation of nerve fibres, *dendrites*, are connected to a cell body, *soma*, within which is located a nucleus. A single long fibre, the *axon*, leaves the cell body which ends by repeatedly dividing. The terminating points of the divided axon form transmitting connections to the dendrites of other neurons or connect directly to the neurons via *synaptic junctions* or *synapses.*

Signalling from one neuron to another is a complex chemical process with chemicals released from the sending side of the synapse. The effect of these chemical releases is to alter the electrical potential within the cell body. If the cell potential reaches a given level the neuron is activated releasing a fixed strength and duration signal along the axon to other neurons. After the cell has fired a recovery period follows before the neuron is able to fire again. (For a more comprehensive explanation of the biological operation of a neuron a biological/medical text should be studied eg. Gray's Anatomy). Individual cells and interconnections are limited in the task which they can achieve, but the collective behaviour of these structures of biological formations performs a useful task in the embodying organism. Conservatively it has been estimated that there are at least 10^{11} neurons in the human brain with 10^{14} interconnections ie. 10^{3} synapses for each neuron.

Given the above rudimentary description of a neuron's behaviour two main approaches can be adopted for the study and development of ANNs. One approach is to study, model and possibly build analogous devices as accurately as possible. The second is to draw upon ideas from actual systems and develop simple processing element exemplar within a massively parallel architecture. The former approach is normally adopted by biologists and psychologists in order to determine the functioning of the brain and nervous system. The latter approach is usually followed by engineers in pursuit of a system which will perform a computationally useful task. This is the method that will be followed while still remembering the inspiration for the ideas.

A final few points should be made clear about NNs, that is a NN is not a static entity. The strengths of interconnections vary with time, new ones are formed and old ones may decay away. Due to the large quantity of parallelism there is redundancy built into the system and a level of fault tolerance is available. Rather than being explicitly programmed a NN evolves to perform an action by learning and adaption. Thus, given that the network changes through damage or the network has to increase its functionality it is able to adapt to the new situation. It is necessary therefore to study and develop learning/training algorithms for any network created to enable it to be taught how to perform a task or tasks.

Why study and develop ANN at all? What benefit can they offer beyond a traditional von Neumann architectured machine? What task or tasks could they be used to perform? Hopefully a more complete reason for the study of ANNs will become apparent by answering the latter two questions.

Benefits of ANN are their potential robustness and gradual degradation in performance if an area of the network becomes damaged. Within a traditional computer a failure in a processing section is catastrophic in terms of system performance, this is not necessarily the case with a NN. A von Neumann machine must be explicitly programmed to perform a task. Even with the use of a high level programming language this may not be a simple operation for a complex task or the genre of operation which a NN is actually accomplished at. Certainly for rapid exact algorithmic or mathematical operations a traditional computer is excellent but this is not the case for noisy, inexact information processing.

A NN can perform as a *classifier* where the task of a classifier can be divided into the following three categories.

Traditional Classifier. A NN can be used to identify a class to which an input is most appropriate, eg. to classify types of vehicles as to whether they are cars, vans or bicycles. The difference a NN classifier exhibits from a statistical classifier is that it is adaptive and is able to take into account new information as opposed to processing all training data before being used with new data. NN may be non-parametric and make fewer assumptions about a data set's information distribution than a traditional classifier.

Content-Addressable or Associative Memory. These are similar operations. In Content-Addressable Memory (CAM) data are mapped to an address, whereas with Associative Memory data are mapped to data. In this mode of operation a NN may be used to recall a more complete pattern for a piece of input data eg. a partial image of a character can be used to reconstruct the entire character or a telephone number will lead to the retrieval of the name and an address associated with it.

Vector Quantiser or Feature Extractor. In this situation a NN may not be provided with any *a priori* information about a data set but is taught to cluster the information as it sees fit by the extraction of information it considers relevant. These NN could be used in signal transmission to reduce the information which must be sent without losing the clarity of the message. Similarly in data compression they may be used to extract only pertinent information for storage.

Already it has been stated several times that much of the interest and power of NNs is the ability they have to adapt and learn from the data presented to them. The two global classes of training available are *Supervised Learning* and *Unsupervised Learning.* These two classes can be sub-divided into learning *structural* information or *temporal* information.

Supervised Learning. In this case the desired output from the NN is known for each input and is used to improve the NN output performance. This improvement can be by direct comparison of each desired output and actual output or by the use of a performance signal which indicates how satisfactorily a NN has performed for the given input. This case is often referred to as *Reinforcement Learning* or *learning with a critic,* whereas overall Supervised Learning can be referred to as *learning with a teacher.*

Unsupervised Learning. This system has no external teacher to guide a NN response. The network is allowed to form its own internal clusters of information. Unsupervised Learning can be called *self-organisation.*

The two sub-categories of structural and temporal learning are described as follows. With structural learning a stable attractor exists for each input which will be learned. For temporal learning the output could be a sequence or series of patterns. Whether or not the input is structural or temporal will be problem specific.

2.2 Basic Processing Element Model

The structure of the basic artificial neuron can be traced back to the work of McCulloch and Pitts, 1943, [3]. They proposed that a model neuron would be either on, firing, or off, not-firing, based upon the weighted sum of inputs exceeding a threshold value. For an n input neuron where x_i is an input, w_i is the associated weight the response N_{out} is such that

$$\sum_{i=1}^{n} x_i w_i \geq T \Rightarrow N_{out} = 1$$

else

$$\sum_{i=1}^{n} x_i w_i < T \Rightarrow N_{out} = 0$$

where T is the threshold at which the neuron is activated. To make the threshold of activation easily variable it can be treated as another weighted input, $x_0 w_0$, the input value of which, x_0, is always unity.

$$\sum_{i=0}^{n} x_i w_i \begin{cases} \geq 0 & \Rightarrow & N_{out} = 1 \\ < 0 & \Rightarrow & N_{out} = 0 \end{cases}$$

This basic neuron architecture of McCulloch and Pitts can be graphically summarised as in Figure 2.2.

The step threshold function is only one of several *activation functions* which an artificial neuron may have. Other common neuron activation functions are the linear, clipped linear and sigmoidal function as illustrated in Figure 2.3.

Many different ANN models have been developed including the Perceptron, Multi-layer Perceptron, Kohonen Self-Organising Feature Map, Hopfield Net, Boltzmann Machine, Bidirectional Associative Memory, Adaline, Madaline .... Each network structure exhibits its own style of functionality, structure and learning technique. In order to appreciate the diversity of the subject and to gain an insight into the operation of ANN several of the above models will be discussed.

2.3 Single-layer Perceptron and Multi-layer Perceptron

The term perceptron was coined by Rosenblatt for his implementation of the McCulloch & Pitt style neuron. Rosenblatt studied this form of artificial neuron extensively as summarised by himself [4] and more simply by Simpson [5] or Hertz *et al* [6]. These two styles of network which are of interest are both feedforward networks, ie. all interconnections between neurons are in a forward direction only with no connections feeding backwards to previous neurons and no connections feeding across to neurons at an equivalent depth in the network, both are feasible in more sophisticated configurations. The Single-layer Perceptron (SLP) is the most basic network but it is still able to perform simple pattern recognition tasks. Training may be achieved by the Perceptron Convergence Procedure. More complex pattern recognition may be achieved using the Multi-layer Perceptron (MLP) which after the development of the Backpropagation algorithm could also be successfully trained.

2.3.1 SLP and the Perceptron Convergence Procedure

A single perceptron computes a sum of weighted inputs which after subtraction of the threshold, T, passes the resultant through a step threshold activation function to produce either a 1 or -1 as its output. The activation function is the sgn function. The perceptron may be considered to respond to one class of inputs with a 1 and to the rest with a -1. If the perceptron output is y then

$$y = \operatorname{sgn}\left(\sum_{i=1}^{n} x_i w_i - T\right)$$

$$= \begin{cases} +1 \\ -1 \end{cases}$$

Once again the threshold can be subsumed into the summation as an input x_0 which is always unity. A perceptron can be seen to form two decision regions which in a two input case produces a dividing line, for the three input case a dividing plane and in higher dimensional cases a dividing hyperplane. The exact position of this decision boundary is adaptable by adjusting the weights and training the perceptron to respond correctly.

A SLP architecture is illustrated in Figure 2.4. It can be seen to consist of two layers only. The first or input layer acts only to distribute the inputs to each perceptron on the second, processing, layer. The processing layer produces the network outputs.

How can the weights which connect the input layer to the processing layer be adjusted? Rosenblatt proposed the Perceptron Convergence Procedure which will now be described step by step. NB. T has been incorporated as $x_0 w_0$.

1. Initialise all weights, w_i, to a small random value. $0 \leq i \leq n$

2. An input vector **X** and the desired output vector **D** are presented to the network of n perceptrons.

$$\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$$

$$\mathbf{D} = \{d_1, d_2, \ldots, d_n\}$$

$$d_i = \begin{cases} +1 \\ -1 \end{cases}$$

3. Calculate the actual output vector of the SLP **Y** by determining the response of each perceptron.

$$\mathbf{Y} = \{y_1, y_2, \ldots, y_n\}$$

$$y_i(t) = \operatorname{sgn}\left(\sum_{i=1}^{n} x_i w_i\right)$$

4. Adjust the weights according to the following scheme.

$$w_i(t+1) = w_i(t) + \eta[d_i(t) - y_i(t)]x_i(t)$$

$$0 \leq i \leq n$$

η a gain term used to specify the proportion of adjustment required, the adaption rate, $0 < \eta \leq 1$

5. Repeat from step(2) until a satisfactory response is produced from the network for the classes of data.

   It will be seen from step(4) that no weight adjustment occurs if the actual output is equivalent to the desired input, $y_i(t) - d_i(t) \equiv 0$.

The selection of the gain term η is important as it must satisfy two conflicting constraints, that of producing fast adaption for variances in input and the alternative of producing stable weight estimates from past events. The greater η is the quicker adaption will occur but the less stable the adaption will become. Choice of η is very much problem dependent.

Variations on the basic Perceptron convergence procedure can be made by using a continuously valued activation function output from the perceptron rather than the sgn function. This will allow the use of gradient descent techniques for perceptron weight adaption. If an error or cost function is defined for the SLP output e such that

$$e = \frac{1}{2}\sum_{i=1}^{n}(d_i(t) - y_i(t))^2$$

the change in the weight w_i can now be made proportional to the gradient of the error at the present location.

$$\begin{aligned} w_i(t+1) - w_i(t) = \Delta w_i(t) &= \eta\frac{\partial \mathrm{E}}{\partial w_i(t)} \\ &= \eta\sum_{i=1}^{n}(d_i(t) - y_i(t))x_i(t) \end{aligned}$$

The correction in weight value can be made individually leading to

$$\Delta w_i(t) = \eta\delta_i x_i(t) \tag{2.1}$$

$$\delta_i = d_i(t) - y_i(t) \tag{2.2}$$

The equation eq.(2.1) and eq.(2.2) form the *delta rule*, *adaline rule* or *Widrow-Hoff rule* [7]. A more common name and the one most often applied in an adaptive signal processing field is the Least Mean Square (LMS) rule.

The SLP is a very simple NN and as such suffers from several constraints. For a perceptron to be able to make a decision the two distribution domains must be linearly separable, it must be feasible to form a dividing plane between the two domains. For example the two input AND function is linearly separable whereas the two input exclusive-OR, XOR, is not Figure 2.5. The XOR problem is the simplest case of a parity decision problem, the more general class of which is discussed by Minsky & Papert [8]. If the domains are not linearly separable no stable decision can be made and the boundary will alternate for the different input sets. If the classes are too close together it may prove difficult for a decision boundary to be formed, but given that a set of weights for the desired association does exist it has been proved by Minsky & Papert [8] and Hertz *et al* [6] amongst others, that the Perceptron Convergence Procedure will find them in a finite number of iterations. The drawback here for the SLP is thus the potentially long learning time. Due to the SLPs simple decision nature they are poor at generalising a solution.

Before proceeding forward to describe the more powerful MLP systems much emphasis has been placed upon the work of Minsky & Papert for quashing enthusiasm for the ANN within their book Perceptrons. In a revised and updated 3rd edition they argue forcefully that their intention was to highlight considerations which must be borne in mind when evaluating neural systems and their classification potential through examples of hard learning problems, eg. the N-input parity problem or the determination of connectedness. It would be fair to say that no adequate learning algorithm existed at the time for training multiple layered networks. These problems have subsequently been resolved independently by several researchers as described in the following section on MLPs.

2.3.2 MLP and Backpropagation

As the name suggests the MLP is an extension of the SLP to create a network of more than one layer of perceptrons. If the perceptrons have a continuously valued non-linear activation function many of the limitations of the SLP can be overcome. It is this type of activation function which provides the network with the ability to perform more complex tasks. If the processing elements had linear activation functions then the MLP can be demonstrated to be reduced to a SLP. The problem with the MLP originally was the ability to adjust the weights of all perceptrons in a coherent fashion to improve the network performance. The advent of the Backpropagation algorithm has removed this hurdle.

Before describing the Backpropagation algorithm it would be wise to first of all specify a naming and numbering convention for the MLP. An MLP consists of a number of layers of perceptrons as illustrated in Figure 2.6. There are three types of layer within an MLP, input, hidden and output layers. The first layer, the *input* layer, acts purely as a distribution layer, each node supplying signals to processing elements in the following layer. No processing takes place at this level. The last layer, the *output* layer, receives all the inputs for its processing elements from within the network and passes the results back out to the environment. Between the input and output layers there are one or more *hidden* layers, so called because they have no external connections to the environment. Signals are received from the previous layer, processed and outputted to the following layer. Due to the isolation of hidden layer processing elements they are often the most difficult to analyse and adapt. An MLP will be specified by the number of hidden layers plus the output layer that it contains and by the number of neurons in each layer. This is based upon the fact that processing only occurs in these layers and neurons. Hence, Figure 2.6 is a three layer MLP of configuration 4–3–3 with three inputs and three outputs.[1]

Being able to specify a network is clearly one consideration, another is how is the number of layers determined? and how the number of perceptrons are determined for each layer? Quite obviously the number of nodes for input and output will be determined by the required connections to the environment, for hidden layers the task is not so simple. Lippman [9] highlights how the decision regions are constrained by the various number layered networks from the SLP upto the three layer MLP. In theory an arbitrary complex decision space can be created by a three layer MLP, more layers may be used to aid in the decision region formation. The number of perceptrons in a hidden layer must be sufficient to form decision regions that are as complex as required but no more. Too many perceptrons may cause the network to overclassify ie. its response is too highly tuned towards a particular set of inputs rather than a general class of inputs, the network therefore has difficulty generalising.

For a more formal analysis of the number of hidden layer perceptrons required and their ability to divide the solution space the work of Mirchandani & Coa [10], Huang & Huang [11] and Makhal *et al* [12] should be consulted. These papers unfortunately place constraints upon the MLP configuration to obtain their results. In the general case they may not be so applicable. They do illustrate the complexity of the analysis necessary for even the simplest of networks.

Given that a network has been formed and it is possible to alter the weights for the interconnections, what method should be used to determine how to vary the weights? For the SLP the Perceptron Convergence rule exists for producing the correct output or there are the gradient descent technique variations, delta rule etc. for minimising the error between actual output and desired network output. By extension of this gradient descent approach for minimising a cost function several researchers have developed the same appropriate algorithm commonly known know as Backpropagation, Werbos [13], Parker [14] and Rumelhart *et al* [15]. The name is taken from the most recent exposition of the algorithm by Rumelhart *et al.*

Backpropagation is an iterative gradient descent technique with the aim of reducing the difference between the actual and desired output. The technique relies upon each

[1]Caution: Some papers include the input layer in the specification of the size of the network.

processing element possessing as its activation function a continuously differentiable non-linear function. A sigmoidal transform is most often used.

$$f(x) = \frac{1}{1+e^{-x}}$$

$$\begin{array}{rl} x \to +\infty & f(x) \to 1 \\ x = 0 & f(x) \to 0.5 \\ x \to -\infty & f(x) \to 0 \end{array}$$

or

$$f(x) = \tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

$$\begin{array}{rl} x \to +\infty & f(x) \to 1 \\ x = 0 & f(x) = 0 \\ x \to -\infty & f(x) \to -1 \end{array}$$

There follows a step by step description of the backpropagation algorithm as put forward by Rumelhart *et al.*

1. Initialise all the weights w_{ij} to small random non zero values.

2. An input vector $\boldsymbol{X}$ and the desired output vector $\boldsymbol{D}$ are presented to the MLP.

$$\boldsymbol{X} = \{x_1, x_2, \ldots, x_{N-1}\}$$

$$\boldsymbol{D} = \{d_1, d_2, \ldots, d_{M-1}\}$$

3. Forward propagate through the MLP from the input layer to the output layer. The response for each layer is calculated and fed into the following layer until an output $\boldsymbol{Y}$ is produced.

$$\boldsymbol{Y} = \{y_1, y_2, \ldots, y_{M-1}\}$$

4. Adapt the weights for each layer starting at the output layer and backpropagating the adjustment through the hidden layers.

$$w_{ij}(t+1) = w_{ij}(t) + \eta\delta_j x'_i$$

$w_{ij}(t)$ weight for hidden node i or input node i in preceding layer to node j in current layer at time t.

x'_i output of node i in preceding hidden layer or actual input value.

η gain term which determines the degree of adaption to weight.

δ_j a correction measure based on the error between the desired and actual response. This is calculated differently for the hidden layer and the output layer.

Output layer The desired response is d_j while the actual response is y_j.

$$\delta_j = y_j(1 - y_j)(d_j - y_j)$$

Hidden layer There is no known desired response therefore an expected response is inferred from the following layer.

$$\delta_j = x'_j(1 - x'_j)\sum_k \delta_k w_{jk}$$

k is for all neurons in the layer after node j.

5. Repeat this procedure from step (2) until the network performance is acceptable.

The above listed basic algorithm suffers from the fact that it can take a long time to converge and also that it is possible for the system to become caught in a local minima of the solution space rather than the global minima. One of the most useful and widely implemented techniques to improve this basic algorithm is to include a *momentum* term α at step (4). The momentum takes into account the amount by which the weight changed on the previous pass through the algorithm. The improved weight update equation is

$$w_{ij}(t+1) = w_{ij}(t) + \eta\delta_j x'_i + \alpha\Delta w_{ij}$$

$$\Delta w_{ij} = w_{ij}(t) - w_{ij}(t-1)$$

$$0.0 \leq \alpha \leq 1.0$$

The reasoning behind the use of the momentum term is that, as the algorithm changes the weights downwards towards the global minima, the momentum term will provide averaging across the different input/output pattern pair sets presented. If local minima occur the momentum term should enable the algorithm to pass through them more easily without being trapped. NB. For $\alpha = 0$ the update equation reduces to that of the basic backpropagation algorithm.

As different values of η and α may be optimal at different points it has been proposed to make them adaptable eg. Vogl [16] and Hertz *et al* [6]. One such scheme is to vary η based upon the effectiveness of η at reducing the error. If η did not cause a reduction in error the weight adjustment η is too severe and should perhaps be reduced. Conversely if several updates have been made which cause the error to reduce, η may be increased as the adjustment that it causes is too conservative.

$$\Delta\eta = \begin{cases} +a & \text{if} \Delta E < 0 \ \text{consistently} \\ -b\eta & \text{if} \Delta E > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

eq.(2.3) is a proposed gain adjustment scheme, the gain is improved by a constant step a if consistent improvements in the network performance are made, while a proportional deduction of gain occurs for poor network performance. It has been suggested that α should be set to 0 when the gain is reduced and reset to its original value when improvements in gain are made. The reasoning for this step is that the momentum term takes account of prior learning experiences Δw_{ij}, thus when the change in network error ΔE is positive the general direction of weight change should reverse, a process which the momentum term opposes.

Other techniques for improving the scope and performance of the basic backpropagation algorithm include Scalero & Tepedelenlioglu's [17] system for minimising the mean-squared error between the actual and desired outputs with respect to the inputs to the non-linearities. Training in the complex domain can be achieved by using Complex Backpropagation which may take several variant forms, [18, 19, 20, 21].

The MLP and backpropagation discussed so far are a restricted form of the general class of feedforward networks. More generally the output of a neuron is able to feedforward to any neuron in any layer of the network. It is unnecessary to connect the output of a neuron to all the inputs of the neurons in the following layer. This relaxation of conditions from the fully connected MLP lead to much of the fascination with the structure of ANNs. If a link or a neuron fails it may be possible to readjust the weights to restore the performance of the network. The system has fault tolerance and the ability to re-adapt. If the performance of the network is affected it will most likely be a gradual deterioration rather than a catastrophic failure of the whole system.

It can be seen that overall the backpropagation algorithm is quite numerically intensive requiring a lot of information to be passed both forward and backwards. At each neuron many calculations must be performed and a record of previous weight conditions maintained if the momentum term is to be utilised. Backpropagation is not suited to direct implementation in hardware upon a specialised platform which operates on-line. Usually, learning, training and adaption are performed off-line and the learned weights programmed into hardware which is to run the network, whether that be a conventional architectured machine or a more highly specialised piece of hardware for running a NN.

2.3.3 MLP and Backpropagation Implementation

To acquire an understanding of the problem of implementing an MLP network and to apply the backpropagation training algorithm in software a simple simulator was produced. It should be noted that many sophisticated and respected NN simulators exist both commercial eg. NeuroProII or 'public domain' eg. Xerion or Migraines/Aspirin. It was felt that benefit would be gained by producing a simple demonstrator with which to experiment.

The simulator enabled simple networks of up to five layers and forty neurons per layer to be specified. Configuration of the simulator is controlled by a setup file `setup.mlp`. An example of the file `setup.mlp` is shown in Figure 2.7. The format of the file is slightly terse and the actual specification of the network is not to the standard described in the previous section, this was to simplify coding. The file terms are explained as follows:

`layers` the total number of layers in the network (input, hidden and output)

`neurons per layer` the appropriate number of layers to describe each layer

`training gain` the value of η

`training momentum` the value of α

`tv` the number of training vector combinations $\boldsymbol{X}$ and $\boldsymbol{D}$

`inspect rate` how frequently the RMS error of the network is to be stored in the file `results.mlp`

`training group size` the number of times a training vector pair is to be presented to a network before the next training pair is selected

`epochs` the number of different training vector pairs to be presented

`ip/op` the appropriate number of training vector combinations

The output of the software is an ascii file `results.mlp` which firstly reiterates the network parameters followed by a table of the RMS error of the network against time.

Two standard demonstration problems were investigated using the simulator, the 8–3–8 coder/decoder and the two input Exclusive-OR (XOR). These problems were chosen to validate the literature on the general characteristics of an MLP network.

Encoder/Decoder Problem

The encoder/decoder problem is an `auto-associative` problem in which the network output $\boldsymbol{Y}$ matches the original network input $\boldsymbol{X}$.[2] The aim is for the MLP to find a

[2]In a *hetero-associative* problem the network output Y differs from the network input X.

suitable coding scheme to pass the input pattern through a reduced number of hidden layer neurons back out to the same number of output neurons as inputs. This type of problem may also be referred to as an N–M–N problem where M < N. The difficulty of the learning problem depends upon how much smaller M is than N. Specifically a two layer MLP was used to solve the 8–3–8 encoder/decoder problem. There are eight input/output patterns each with a single input set high in each input pattern and only the corresponding line set high in the output, in fact Figure 2.7 illustrates the eight training vectors. The obvious solution to the 8–3–8 problem is for the three hidden layer neurons to learn the binary codes.

A group of simulation runs were performed with various combinations of η, α, `training group size`, and whether the patterns are presented individually at random or sequentially as a batch. Figure 2.7 is actually a setup file for such a problem, there being eight training vector combinations. The results of these simulation runs can be seen in Figure 2.8 to Figure 2.12.

The first set of runs had zero momentum, $\alpha = 0$, and individual training vector pairs were presented at random, Figure 2.8. It can be seen that increasing the gain term for backpropagation increases the rate of error reduction. However, although for larger gains a faster rate of convergence occurs, the descent is more noisy and the system varies around the convergence point more as it over corrects.

The next two sets of runs had a non-zero momentum term and again individual training vector pairs were presented at random, Figure 2.9 and Figure 2.10. These figures illustrate that increasing the momentum term increases the speed of the error reduction, a combination of relatively large gain and momentum produce the fastest converging results. The two terms cannot be increased continuously or else the system becomes unstable.

Finally for the encoder/decoder case two sets of batched runs were performed as shown in Figure 2.11 and Figure 2.12. In these runs all of the training pairs were passed through the network and the average RMS error for all pairs used as the means of network training by backpropagation. Both figures demonstrate what has already been shown that increased gain or momentum can increase the rate of adaption.

The overall speed of adaption is generally comparable for both the individual and batch methods of pattern presentation but the batch system produces a smoother RMS error curve and will be a smoother path across the error surface of the system.

The analysis of a cross section of runs for many gains and momentum combinations reveals that the limiting values for both are interdependent. In general, the larger the value of one parameter the lower the limit of its counterpart. A possible solution to this interdependence and noisy convergence is to use adjustable values. Initially large values for both parameters are selected, first the momentum term is reduced and later the gain term. In this way a rapid descent of the error surface could be achieved initially, but as a solution is reached the noise in the gradient following will be reduced first as the momentum and then as the gain is reduced.

XOR Problem

The XOR problem is a hard learning problem so called because the input/output relationship is not linearly separable, as illustrated in Figure 2.5. The XOR problem is the simplest form of the more general N-input parity problem given by Minsky and Papert, [8]. For an XOR there are two inputs and one output. The output is high if either one or other of the inputs is high, but not both. The more general N-input parity problem is such that the single output is high if either an odd or even number of the N inputs are high depending whether odd or even parity is required.

For these tests a fully connected two layer MLP with two hidden layer neurons and one output neuron is used. It should be noted that Rumelhart *et al* [15] demonstrate a simplified feedforward network solution to this XOR problem using the network of Figure 2.13. In this case though it can be seen that connections are utilised which skip the intermediate hidden layer allowable in a general feedforward structure but not in our restricted case of an MLP.

A group of simulation runs were performed with various combinations of gain and momentum. The results of these simulation runs can be seen in Figure 2.14 to Figure 2.16. It can be seen that similar characteristics are exhibited as for the 8–3–8 encoder/decoder problem in that larger values of gain or momentum produce faster rates of error reduction. However, with this problem it can be seen that, within the duration of the runs, the network did not always converge to a satisfactory solution, Figure 2.14 for $\eta = 0.5$ and $\alpha = 0.0$, or Figure 2.16 for $\eta = 0.7$ and $\alpha = 0.4$. It was found that often re-initialising the weights at random values enabled the system to converge for the same system training parameters. For these runs it can also be seen that the rate of error reduction once it does start to occur is rapid.

2.4 Kohonen Self-Organising Feature Map

Supervised learning as demonstrated by the Multi-Layer Perceptron is only one form of learning. It is not always necessary to have a formal teacher to train a neural network. Teuvo Kohonen has developed the self-organising neural network in his work, [22, 23, 24]. This type of network performs its classification and learning in an *unsupervised* manner. No explicit tutorial set of inputs and outputs is required.

The biological origin of the Kohonen Self-Organising Map is the competition exhibited within sectors of physiological neural systems and the resulting spatial organisation of response. There is direct evidence of the localisation of functions inside the brain. Within localised areas maps exist for variations of a given type of stimulus. For example, an area of the brain responds to sound stimuli, but slightly different sections are excited for different notes.

The Kohonen network operates on a winner takes all policy for the neurons. Each neuron receives identical inputs. Neighbouring neurons in the network compete in their activities by mutual lateral interaction. Pattern detection of the inputs occurs as the neurons adaptively form specific feature detectors, each neuron becoming a separate decoder. The format of the neuron is different to that of the perceptron. The neuron whose weights most closely resemble the input vector is said to be the active neuron and produces a response. The neuron with the active response has its weight values for its inputs adjusted towards the stimulus to improve the response, while other input weights in the net are decreased or left alone. Rather than adjust the values for only one neuron a response neighbourhood structure may be used in which nearest neighbours of an active neuron also have their weights adjusted in favour of a response for the given input vector. Gradually the size of the neighbourhood is reduced as is the degree to which the neuron weights are changed. Types of neuron neighbourhood maps are illustrated in Figure 2.17 and Figure 2.18.

It has been stated above that all neurons receive that same inputs. This does not strictly have to be the case. Kohonen originally proposed the use of a switching or relay network between the network inputs and neuron inputs. Each neuron received a set of signals from the environment which may not be identical but are coherent. It was demonstrated that self-organisation would still occur provided the input events to the neurons are uniquely determined by the input events to the network. Using the Kohonen training algorithm, self-organisation of a set of signal values is only possible if the relationship between signals is simple. For practical applications preprocessing will often be necessary to form a simple association, eg. for image processing.

2.4.1 Training

Unlike feedforward networks, such as the MLP presented earlier, no explicit response is required from the network. Input patterns are presented to the network during training to enable neuron responses to group themselves into areas of similar action.

The unsupervised training algorithm for Kohonen Self-Organising Feature Maps may be described as follows.

i) Initialise all weight values to small random values. Often the weights are normalised for improved network performance.

ii) An input vector, $\mathbf{X}$, is presented to the net.

$$\mathbf{X} = \{x_0, x_1, x_2, \dots, x_{N-1}\}$$

iii) Calculate the distances between the input vector and the weight vectors for each neuron.

$$d_j = \sum_{i=0}^{N-1} (x_i(t) - w_{ij}(t))^2 \tag{2.4}$$

where

$\boldsymbol{d_j}$ distance between input and output of neuron j.

$\boldsymbol{x_i(t)}$ input to node i at time t.

$\boldsymbol{w_{ij}(t)}$ weight for input node i to output node j at time t.

iv) Determine the node j^* with the minimum value of d_j. This is the active neuron.

v) Improve the weights of neuron j^* such that its response for this type of input is a closer match, ie. d_j is smaller. Enhance the weights values for all neurons in the designated neighbourhood by the following system.

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t) - w_{ij}(t)) \tag{2.5}$$

$$0 \leq i \leq N-1$$

$$j \in NE_{j^*}(t)$$

$\boldsymbol{\eta(t)}$ training gain at time, t, $0 < \eta(t) < 1$

Note the similarity between the perceptron weight updating, eq.(2.11), and the Kohonen weight updating, eq.(2.5).

vi) Adjust the training gain, $\eta(t)$, if required. Training gain should be reduced monotonically with time.

vii) Adjust the size of the neighbourhood if required. The size of the neighbourhood should be reduced monotonically with time.

viii) Repeat training from step (ii) with a new input pattern until a satisfactory response is achieved from the Kohonen Self-Organising network.

In steps (vi) and (vii) of the training algorithm, how would it be best to vary the training gain, η, and the size of the neighbourhood?

Taking the size of the neighbourhood first. A wide neighbourhood should be specified initially to provide general ordering of the neurons in relation to the inputs. The size could be up to half the total number of neurons. As learning progresses the area should be reduced to produce improved local ordering. It may finally occur that only one neuron is adjusted for a given input. The specific method of area reduction is not particularly important, linear, exponential or proportional to time are all successful. Due to neurons being discrete entities the reduction will need to be quantitised.

Training gain may be adapted in a similar fashion. The gain is specified to be between zero and unity. For values close to unity the adjustment of weights is large and may be used to provide general ordering. For values close to zero the adjustment will not be as significant to the reordering of the network, but more towards the fine tuning of neuron responses. Again it is not significant which particular method is used for reducing the gain. Unlike the neighbourhood size, adjustment of the gain will be continuous.

The two ideas of reducing the influence of training gain and neighbourhood size may be combined in the use of a training gain that is variable with the distance from the active neuron, Figure 2.19.

The active neuron has the most adaptation, as one moves towards the outer layers of the neighbourhood the gain is reduced. A bell shaped gain centred on the active neuron is often used. As learning progresses it is still necessary to reduce the overall gain and neighbourhood size with time.

It has been found that the maps formed by Kohonen networks have the following convergence properties

i) representation of the divisions of the data amongst the inputs are formed along the most pronounced dimensions.

ii) preservation of the neighbourhood relationship between inputs.

iii) transform regions of input domain which are more frequent to larger regions of the output domain with greater detail and *vice versa.*

2.4.2 Kohonen Self-Organising Map Implementation

To acquire an understanding of the problem of implementing a Kohonen Self-Organising feature map and to apply the learning rule in software, a simple simulator was produced as per the MLP, §2.3.3. The basic algorithm implementation was straightforward, but addition of varying learning rates, neighbourhood sizes and the input/output of information proved more time consuming.

A simple problem was addressed, that of ordering two-component input vectors , (x, y) where $1.0 \leq x \leq 10.0$ and $1.0 \leq y \leq 10.0$. A two dimensional array of two input neurons was used. An ideal mesh can be visualised for uniform response, Figure 2.20 shows a 10 by 10 ideal mesh. One corner of the Kohonen layer responds to input vector $(1, 1)$ and the diagonally opposite corner responds to $(10, 10)$.

For an arbitrary input vector the neuron with the closest match, minimum value of d_j, fires. For input $(5.1, 7.8)$ the neuron $(5, 8)$ in the mesh fires. No orientation is specified for the mesh output, so in Figure 2.20 $(1, 1)$ could equally be the top right, bottom left or bottom right after training but with $(10, 10)$ always diagonally opposite to preserve the neighbourhood relationship between inputs.

The algorithm of §2.4.1 was adopted with a neighbourhood style of Figure 2.17. The weight vector values were set to random values near the mid range of the training space, $(5.0, 5.0)$. Uniformly distributed random vectors were presented to the network with different values of η_0, neighbourhood size and the rate of their reduction.

By displaying the mesh created by the neuron weights at successive intervals the organisation of the network can be viewed graphically. Initially for large value of η_0 and large neuron neighbourhoods the mesh dynamics are large, large changes in the mesh layout occur as general ordering occurs. The neurons orientate themselves towards an appropriate topology. Once topologically correct the refinement of the weight values occurs.

The concept of reducing η and the neighbourhood size can be considered as the amount of energy or heat which the system possesses. At high values much movement of weight values and hence mesh layout are possible due to the high energy of the system. The reduction of η and neighbourhood size may be likened to a cooling process enabling the system to settle into an ordered state.

The series of figures, Figure 2.21 to Figure 2.24, illustrates the organising process of the Kohonen layer. It can be seen how the network organisation settles down with increased number of pair presentation. In these figures the data in the (x, y) pair are uniformly distributed throughout the input space. Due to the simplicity of the input set it is difficult to demonstrate the principle that the division of data amongst inputs has occurred along the most pronounced dimensions. A two-dimensional layer is being used to divide a two input vector. The neighbourhood relationship is preserved in the Kohonen Layer.

The uniform distribution of data does not allow the demonstration of the transfer between domains, ie. the areas of the input domain which are most frequently excited are mapped to larger regions, more neurons, in the output domain. To verify that this occurs the distribution of the (x) component of (x, y) was changed to a normal distribution. The normal distribution is centred at the middle of the range.

Figure 2.25 illustrates the effect that this has on the output domain. Firstly, the spread in the x direction is reduced, for the uniform case the distribution of x and y was the same. Secondly, in the centre of the distribution more neurons are active hence the outputs are closer together produce greater detail. All input weights are adjusted when neuron weight values are improved; this has the effect of causing the y dimension to be drawn in at the top and the bottom.

Kohonen, [22], notes several effects within these feature maps.

Magnification factor which is basically a restatement of the network property that regions of the input domain which are most frequently excited will map to the most neurons in the output domain to produce the greatest resolution.

Boundary effect which forms since the training of neurons occurs in neighbourhoods, those neurons which are near to the edge will suffer an effect due to not having the same number of neurons with which to interact. In general this will cause the map to contract and pull away from the edges of the output domain.

Pinch/Collapse/Focusing Phenomena are all related since they are believed to be caused by the interaction between neurons being incorrect, ie. the wrong parameters for neighbourhood size and strength of interaction. Pinch occurs when the neighbourhood is too small, and means that the distribution of neuron response does not spread out across the entire output domain. Collapse can occur when the neighbourhood is too large and results in many neurons having basically the same output. Focusing can occur if the neighbourhood interaction is too weak, in which case one or two elements take over responding to virtually every input vector presented to the network.

It is found that a balance exists between the rate of reduction of η and η_0. Similarly for the neighbourhood size. Too large a value of η_0 or too slow a rate of reduction and the network takes a long time to settle down and organise into a sensible state. Too small a value of η_0 or too fast a rate of reduction and the network cannot unravel itself into an ordered condition, but remains contorted. Despite these potential pitfalls and the undesirable effects above the Kohonen Self-Organising feature map has been found to be remarkably robust at learning this data set. This must be qualified by stating that the data are not particularly complex and are suitably conditioned to the output domain.

2.5 The Hopfield Network

In the previous sections of this chapter the NN structures of the MLP and the Kohonen Self-Organising Feature Map were reviewed and investigated. In this section a brief discussion of the Hopfield Network is conducted. This NN structure was first presented by Hopfield in 1982 and 1984, [25, 26]. The Hopfield Network is worthy of review because:

1. the network exhibits Associative Memory properties ie. given part of a piece of input data the network is able to more fully recall the entire piece of information.

2. in its original form, the network operates asynchronously.

3. the simple nature and operation has led to its use as the basis for the investigation of hardware implementations of NNs as pointed out by Murray *et al*, [27].

4. the network can be adapted and used to solve a difficult but well designed optimisation problem, including the Travelling Salesman Problem, [28, 29].

2.5.1 Architecture and Operation

The basic architecture of a Hopfield Net is illustrated in Figure 2.26. From this diagram it can be seen that this NN consists of a single layer of neurons which are fed both from the inputs to the network and from every output of the network except their own. The input connections are used to simply load the network. A form of recurrence or feedback exists in the network through the strong coupling of connections from output to input. The aim of the connections is to provide mutual excitation if associated connection weights are positive and inhibition if connection weights are negative.

In the original format each neuron had a step response function with an output value which could be classed as -1 or +1, a sgn function. Given that a set of neuron weights has been determined, to operate the Hopfield Net the following procedure is followed,

1. Load the Hopfield Net with the initial values of the input pattern, $\boldsymbol{X}$

$$y_i(0) = x_i$$

$$0 \leq i \leq N-1$$

2. Update each neuron, j, output according to the following rule

$$y_j(t+1) = \text{sgn}\left(\sum_{i=0}^{N-1} w_{ij} y_i(t)\right) \tag{2.6}$$

   The update method of the neurons given by Hopfield is asynchronous as this is more akin to the way the brain operates. The asynchronous update may be implemented in one of two ways:

   (a) at each time step select a neuron, j, at random to be updated and apply eq.(2.6).

   (b) each neuron independently updates by using eq.(2.6) with respect to a given probability per unit time.

   As Hertz *et al* [6] point out, the former is best suited for the simulation of Hopfield Nets allowing central control, while the latter is more appropriate for hardware implementations. Both methods equate to the same principal of update but with a different distribution in time.

How are the weights, w_{ij}, initially determined for a Hopfield Net? Rather than a training algorithm as per the above two previously discussed systems, the neuron interconnection weights are initially calculated and fixed within the network. The mathematical format for calculating the connection weights as given by Hertz *et al* will be briefly outlined.

Consider first a single pattern to be held within the network, $\boldsymbol{P} = \{p_0, p_1, p_2, \cdots, p_{N-1}\}$. For the Hopfield net to be stable then

$$p_i = \text{sgn}\left(\sum_j w_{ij} p_j\right)$$

The updating equation of eq.(2.6) will produce no change.

$$\Rightarrow w_{ij} \propto p_i p_j$$

The proportional constant may be taken to be $\frac{1}{N}$ with N the number of neurons in the network.

$$w_{ij} = \frac{1}{N} p_i p_j$$

If a few of the initial values entered into the network are incorrect, the overall summation at a node will swamp the errors producing the desired pattern; after the network has been allowed to update itself over several time steps, the network relaxes.

The expansion to storing many patterns within the Hopfield Net is to allow the superposition of terms for each pattern such that

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{Q} p_i^{\mu} p_j^{\mu}$$

Q is the total number of patterns to be stored in the network. NB. The weights of the Hopfield Net are symmetric, $w_{ij} \equiv w_{ji}$.

Overall this weight setting rule is known as the 'generalised Hebb rule' due to its closeness to the proposal by Hebb, [30], regarding the interaction of synaptic strengths in the brain due to experience. Hebb actually wrote:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

The Hopfield Net as an associative memory, or content addressable memory, has two main limitations. Firstly, the total number of patterns which may be stored is small compared to the number of network connections. The attempted storage of too many patterns within the net may cause the network to relax to spurious patterns unlike any of its stored patterns. The second major limitation is that if two patterns show too many bits in common to one of the other patterns, the pattern may be unstable such that the net relaxes to the other pattern with which the original input shares many common bits. Orthoganilisation procedures have been specified to ameliorate the second of these drawbacks.

In general, when operating, a Hopfield Net relaxes to the stored pattern which is the closest with respect to its Hamming distance from the actual input.

2.6 Boltzmann Machine

The last specific NN structure which will be reviewed is the Boltzmann Machine developed by Ackley *et al* [31]. Discussion and descriptions are also given by Rumelhart *et al* [15] and Hertz *et al* [6]. The Boltzmann Machine has much in common with the Hopfield Net previously described in §2.5, in that it extends many of the principles to a multiple layer architecture if required. As with the Hopfield Net, the processing elements are in one of two states, either on or off, however which state a neuron adopts is probabilistic. Similar to the Hopfield Net links between processing elements are symmetric. Any element, i, which is connected to an element, j, has a weight associated with the link w_{ij}; there is an equivalent connection from j to i of value w_{ji}.

A review of the Boltzmann Machine is worthwhile since, as has already been stated, it can be considered an extension of the Hopfield Net. Secondly the neurons operate stochastically and have a stochastic output, yet their collective behaviour can be trained to perform a coherent and computationally useful task. Thirdly the neurons operate in a stochastic manner upon signals which are deterministic, in §6 an artificial neuron is examined in which the reverse is the case, the neurons operate in a deterministic manner upon signals which are stochastic.

Ackley *et al* demonstrated the ability of the Boltzmann Machine using the 4–2–4 encoder/decoder problem which has been used earlier to assess the MLP, §2.3.3.

2.6.1 Architecture and Operation

The basic architecture of a Boltzmann Machine consists of a network of interconnected neurons. It is not necessary for each neuron to be connected to every other neuron and, due to the bidirectional nature of the connections, it is not a feedforward only network, as with the MLP.

A general arrangement of neurons as shown in Figure 2.27 is thus achieved. The main constraint is that the neurons of the network can be divided into two classes visible and invisible. Visible neurons have connections to the outside world, while hidden neurons are simply connected to other neurons. As has been already stated, the neurons are stochastic ie. the output adopts a value of 1 or -1 according to the following rule

$$\begin{aligned} S_i &= +1 \quad p = g(h_i) \\ S_i &= -1 \quad p = 1 - g(h_i) \end{aligned} \tag{2.7}$$

h_i is the sum of weighted inputs for a neuron i as is usual for a neuron.

$$h_i = \sum_j w_{ij} S_j$$

and the probability $g(h_i)$ is given by the Boltzmann function

$$g(h_i) = \frac{1}{1 + e^{\frac{h}{T}}}$$

where T is a measure of the temperature of the system and h is Boltzmann's constant.

With neurons operating in a stochastic manner, how can network training of a Boltzmann Machine be achieved? Ackley *et al* proposed, and demonstrated, a gradient descent based technique which uses only locally available information to optimise the global network performance. The training is a form of Hebbian learning as described in the previous section. To adapt the Boltzmann Machine it is operated in two configurations, *clamped* and *unclamped.* Statistics are gathered regarding the output values of connected neurons in the two conditions of the network.

In the clamped state the visible neurons are held at their desired values and the network is operated at a given value of T until it reaches equilibrium. A measure of the correlation is made between the output of neuron i and neuron j both being on together. This clamping, stabilisation and measurement process must be repeated for each of the desired network input/output formats or a group of subsets of a content addressable memory format. The clamped correlation values for each of the neuron pairs are averaged.

In the unclamped state, the network is allowed to run without any imposed external constraint on the visible neurons. Again a measure is made of the correlation between the output of neuron i and neuron j once the network has reached equilibrium.

The bidirectional interconnection links are updated according to the following rule

$$\Delta w_{ij} = \frac{\eta}{T}\left(\left\langle\overline{S_i S_j}\right\rangle_{\text{clamped}} - \langle S_i S_j\rangle_{\text{unclamped}}\right) \tag{2.8}$$

$\left\langle\overline{S_i S_j}\right\rangle_{\text{clamped}}$ is the average of the correlation between the outputs of neurons i and j for each of the clamped input conditions.

$\langle S_i S_j\rangle_{\text{unclamped}}$ is the correlation between the outputs of neurons i and j for the unclamped input condition.

η is the training gain, rate of adaption, used for the gradient descent.

T is the temperature at which the system is operated. As training of the system progresses the value of T is slowly reduced.

A complete derivation and alternative descriptions of the Boltzmann Machine training procedure can be found in the previous references, [15, 6, 31]. It is interesting to note that due to the stochastic nature of operation of each neuron a weight change may be in the wrong direction thus enabling the system to avoid local minima.

When operating a Boltzmann Machine in software it is usual to select a neuron at random for output update based upon eq.(2.7). For the system to reach equilibrium at a given temperature, in a clamped or unclamped condition, can take some time. Often the speed of reaching equilibrium can be increased by approaching the desired value of T at which a network is to operate through the Simulated Annealing process. The process of Simulated Annealing will be describe more fully in §5.4.

There is clearly a lot of work to be performed in the operation of a Boltzmann Machine which leads to its main drawback; a Boltzmann Machine operates slowly. As Hertz *et al* highlight there are four nested layers of operation:

1. many weights require updating using eq.(2.8).
2. the calculation of $\langle S_i S_j \rangle$ in an unclamped condition and all the desired clamped configurations.
3. attainment of an equilibrium of operation at a temperature T.
4. the network must operate for many cycles with neurons selected at random for output update via eq.(2.7).

Despite the limitations caused by complexity and slow speed of operation the Boltzmann Machine can and does operate successfully. The network demonstrates that constructive collective behaviour can be obtained in a stochastically operating NN. Finally, it is the first truly recurrent NN which feeds information both backwards and forwards via its bidirectional weights.

2.7 Reinforcement Learning Schemes

Reinforcement learning undertaken by the use of a simple signal transmitted to the neuron elements has taken various forms, and will probably have several more in the future. It differs from other supervised learning strategies, such as backpropagation, which are used for adapting multi-layer feedforward networks. Only a single qualitative response of good/bad performance of the network is provided, an error value. Backpropagation and algorithms of its genre produce a specific response to the network performance, an error vector.

Widrow *et al*, [32], using a single ADALINE, demonstrated 'learning with a critic'. The ADALINE is the artificial neuron developed by Widrow and Hoff, [7]. The ADALINE consists of a sum of weighted inputs passed through a signum, eq.(2.9).

$$x_j = \sum_{1=0}^{N} x_i w_{ij}$$

$$\text{sgn}\ x_j \triangleq \begin{cases} +1, & x_j \geq 0 \\ -1, & x_j < 0 \end{cases} \tag{2.9}$$

Normally within the inputs x_0 will be one set to +1 such that adjusting its weight value will have the effect of adjusting the switching point for the signum function. The learning with a critic architecture is illustrated in Figure 2.28. If the response by the ADALINE is deemed to be good the Critic Switch, b_j, is set to the positive, reward, position. The weights of the ADALINE are adjusted by the Least Mean Square (LMS) algorithm or any other appropriate adaption algorithm to improve the tendency of the ADALINE to produce the same response. However, if the ADALINE performance is bad the Critic Switch is set to the punish position and the weights are adjusted to produce the opposite response.

The above configuration was applied to a temporal problem of playing the card game Blackjack. The ADALINE circuits had the role of a player in the game. The critic response was a good if the game was won by the ADALINE player or bad if the ADALINE player lost. The series of inputs to the ADALINE were the cards as played. The output was whether another card should be taken by the ADALINE player. Only at the completion of the game was the critic involved. The same game was advanced through and each input state rewarded/punished depending upon the overall result of the game. An optimal decision strategy exists for the player's actions in the game of blackjack and it was found that the ADALINE performance improved as more games were played tending towards this optimal decision.

Barto *et al* have worked upon several schemes employing reinforcement learning as the means of training individual or a network of neuron-like elements. The first formulation was the Associative Search Network (ASN) [33, 34, 35]. The second scheme was the Associative Search Element (ASE) and the Adaptive Critic Element (ACE) [36].

The ASN is an associative memory structure. The network learns to output a pattern, **Y**, based upon a given input key, **X**, for and environment, **E**. An association is formed between the key supplied to the network and the pattern output by the network. The network is not explicitly informed of the key/pattern relationship but is trained to maximise a reinforcement signal or performance parameter. The performance of the network is determined by the environment evaluating the pattern output based upon the key input. A full ASN is illustrated in Figure 2.29. It can be seen that the ASN as shown consists of two types of processing elements, the basic adaptive elements, AE, and a single predictor element, PE. The aim of the PE is to aid in the training of the AEs by anticipating the reinforcement/payoff from the environment.

At a given time, t,

$$s_i(t) = \sum_{j=1}^{n} w_{ij}(t)x_j(t)$$
$$y_i(t) = \begin{cases} 1, & \text{if } s_i(t) + NOISE > 0 \\ 0, & \text{if } s_i(t) + NOISE \leq 0 \end{cases}$$

The update of the AE weights uses a previous output of the prediction element.

$$p(t) = \sum_{j=1}^{n} w_{pj}(t)x_j(t)$$

Two update processes are required one for the AEs and one for the PE. For the AEs the update is based upon the reinforcement/payoff received from the environment, $z(t)$, previous AE output values, $y(t-1)$, and previously predicted reinforcement/payoff, $p(t)$.

$$w_{ij}(t+1) = w_{ij}(t) + \alpha[z(t) - p(t-1)][y(t-1) - y(t-2)]x_j(t-1)$$

The update of the predictor weights is achieved by the following expression,

$$w_{pj}(t+1) = w_{pj}(t) + \alpha_p[z(t) - p(t-1)]x_j(t-1)$$

The predictor aims to anticipate the payoff from the environment. The term α and α_p are learning constants determining the rate of learning for w_{ij} and w_{pj} respectively.

The second system investigated by Barto *et al* also had two processing elements, ASEs and ACEs. These two processing elements were used together to learn to control the cart-pole balancing problem. The cart-pole balancing problem consists of a movable cart on which a pole has to be balanced vertically. Normally the cart and pole are restricted to move in a single horizontal direction, Figure 2.30. The pole is maintained in balance by applying impulses to move the cart. This control problem is also known as the broom balancing problem.

The ASE network of Barto's and his colleagues was trained to avoid failure of the cart pole balancing system, ie. the pole fell over or the cart reaching the end of its track. The ASE control and learning system configuration is illustrated in Figure 2.31. This is particularly difficult since failure of the system may occur after a long series of individual control decisions. This system differs from the ASN in that not only is a single control output, y, required but also the status of the environment is fed through a decoder before entering the ASE. The environment is divided into regions by the decoder. For each region a control action is to be associated. The regions are constructed from four parameters, the position of the cart, the velocity of the cart, the angle of the pole and the rate of change of pole angle. These regions are similar to fuzzy regions. The decoder selects just a single region or input to the ASE to be active.

The output of the ASE is given by

$$y(t) = f\left[\sum_{i=1}^{n} w_i(t)x_i(t) + NOISE\right]$$

$$f(x) = \begin{cases} +1 & \text{if } x \geq 0 \text{ (right)} \\ -1 & \text{if } x < 0 \text{ (left)} \end{cases}$$

Due to the random noise term the weight, w_i, only corresponds to the probability that an action will be taken. Learning in this system therefore updates the probability of these actions. The learning rule for the ASE is

$$w_i(t+1) = w_i(t) + \alpha r(t)e_i(t)$$

where

α is the learning constant controlling the rate of change of w_i

$r(t)$ is a real-valued reinforcement

$e_i(t)$ is the *eligibility* of an input.

The eligibility term is based upon the premise that inputs should have a maximum influence a short time after firing and decay to zero afterwards, ie. an input becomes less significant the longer it remains inactive. A simple exponential decay of eligibility may be used.

$$e_i(t+1) = \delta e_i(t) + (1-\delta)y(t)x_i(t)$$

$0 \leq \delta < 1$ determines the rate of decay of eligibility.

This overall system is fairly complex and upon testing the results were found to be poor. This was due to the fact that reinforcement is zero for the majority of the time only taking the value -1 at failure of the system. The more successful an ASE becomes the less frequent the occurrence of a failure signal and the slower the learning.

To improve the performance of the ASE the ACE was added to the configuration, Figure 2.32. The ACE performs a similar function to that of the predictor in the ASN in that the aim of the ACE is to produce a better reinforcement, $\hat{r}$. This reinforcement is for every input to the system and output combination from the decoder, so that reinforcement occurs continuously, not just at failure of the system.

Continuous reinforcement is generated in a similar manner to that of the predictor within the ASN,

$$p(t) = \sum_{i=1}^{n} v_i(t)x_i(t)$$

where

$\boldsymbol{p(t)}$ is a prediction of the eventual reinforcement,

$\boldsymbol{v_i}$ is a weight applied to an input x_i.

The ACE weights are updated by the following scheme,

$$v_i(t+1) = v_i(t) + \alpha_p[r(t) + \gamma p(t) - p(t-1)]\bar{x}_i(t)$$

$$0 \leq \gamma \leq 1$$

$\boldsymbol{\alpha_p}$ is the constant determining the rate of change of v_i,

$\boldsymbol{r(t)}$ is the reinforcement from the environment,

γ is a *discount factor* which will provide for the prediction to decay to zero if no external reinforcement occurs and

$\bar{x}_i$ is a *trace* of x_i value calculated in similar fashion to the eligibility parameter of the ASE.

$$\bar{x}_i(t+1) = \lambda\bar{x}_i(t) + (1-\lambda)x_i(t)$$

$0 \leq \lambda < 1$ which determines the decay rate of $\bar{x}_i$ as per δ for e_i.

The estimated reinforcement, $\hat{r}$, is updated by

$$\hat{r}(t) = r(t) + \gamma p(t) - p(t-1)$$

This system of ASE with ACE was found to be far more satisfactory than the single ASE, due to the continuous reinforcement applied to the ASE.

Although these descriptions of ASE and ASE with ACE have been brief it can be seen that both rely upon a single global signal provided by the environment to improve the performance of the controlling network.

Stochastic learning automatons, as reviewed by Narendra and Thathachar, [37], can employ various reinforcement learning schemes to improve their behaviour in acting with an environment. Figure 2.33 illustrates the link between a stochastic automaton and its environment. As Narendra and Thathachar state, a stochastic automaton has six parts, a sextuple, $\{x, \phi, \alpha, p, A, G\}$.

$\boldsymbol{x}$ is the set of inputs.

$\boldsymbol{\phi}$ is the set of internal states $\{\phi_1, \phi_2, \ldots, \phi_s\}$.

$\boldsymbol{\alpha}$ is the action/output set $\{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ such that $r < s$.

$\boldsymbol{p}$ are the state probability vectors which determine the state chosen at each stage, for a given stage n, $p(n) = (p_1(n), p_2(n), \ldots, p_s(n))^t$.

$\boldsymbol{a}$ is the updating or reinforcement scheme which produces $p(n+1)$ from $p(n)$.

$\boldsymbol{G}$ is the output function which may be either deterministic or stochastic, $G : \phi \longrightarrow \alpha$.

The operation of these learning automatons is to update their action probabilities, $p(n)$, on the basis of the environmental response.

The idea of the reinforcement schemes is simple. When a learning automaton selects an action α_i at stage n, if the input from the environment is not a penalty, $x(n) = 0$, the action probability, $p_i(n)$ is increased while the alternative action probabilities are decreased. If the environment inputs a penalty, $x(n) = 1$, the opposite adjustments are made, $p_i(n)$ is decreased while the other action probabilities are increased. The above can be summarised by the following equations, for when the action at n is α_i the $p_j(n+1)$ terms, where $j \neq i$, are adjusted by

$$p_j(n+1) = p_j(n) - f_j(p(n)) \quad x(n) = 0 \quad \text{nonpenalty}$$

$$p_j(n+1) = p_j(n) + g_j(p(n)) \quad x(n) = 1 \quad \text{penalty}$$

The equation for $p_i(n+1)$ are as follows

$$p_i(n+1) = p_i(n) + \sum_{j \neq i} f_j(p(n)) \quad x(n) = 0 \quad \text{nonpenalty}$$
$$p_i(n+1) = p_i(n) - \sum_{j \neq i} g_j(p(n)) \quad x(n) = 1 \quad \text{penalty}$$

The algorithms and continuous functions $f_j(\cdot)$ and $g_j(\cdot)$ are such that

$$\sum_{k=1}^{r} p_k(n+1) = 1$$

$$p_k(n+1) \in (0,1) \quad \forall \quad k = 1, \ldots, r$$

$$\text{whenever every} \quad p_k(n) \in (0,1)$$

Using the two conditions of non penalty and penalty several variations on the reinforcement scheme may be employed. The updating may be linear or non linear and applied with a combination of reward, penalty or inaction for the non penalty-penalty conditions, ie. Reward-Penalty, Reward-Inaction, Reward-Reward, Penalty-Penalty and Inaction-Penalty.

Note the difference in the approach to learning to that of the ADALINE and ASE formats. Stochastic learning automatons perform updates within the probability space, whereas the others perform updates within the parameter's space based upon the reinforcement signal. As the action selected for an environment is probabilistic, the stochastic learning automaton is able to find the global minimum rather than becoming trapped in a local minima, which can occur for the previous architectures.

2.7.1 Barto Reinforcement Learning

Barto and Jordan, [2], describe a method for performing nonlinear supervised learning upon a multi-layer feedforward network. Instead of the exact solution to the network being used, a qualitative response is created to describe the network's performance. A critic is used to train the network punishing or rewarding the system depending upon its response to inputs. A scalar quantity is fed back through the network to each of the neural elements. In backpropagation an error vector is fed back through the network. The error vector which backpropagation uses contains more information on the differences between the desired output and actual network output.

Barto and Jordan in fact use two variants of an Associative Reward-Penalty or A_{R-P} algorithm an element of stochasticism is introduced into the weight updating mechanism. These two algorithmic variants will now be described. In the following section, §2.8, of this thesis two extensions to these mechanisms are proposed commensurate with the hardware neuron which will be developed.

As already stated, the algorithm operates upon a multi-layer feedforward network. Input signals are applied to the input layer of the network which propagate through to the output layer. Besides the connections to the preceding layer, each processing element also has an input which is permanently at +1, a bias. The input layer processing elements do not actually perform any computation, but act as a distribution point for the signals. Hidden layer processing elements and output layer processing elements generate an output value in different ways.

Output layer elements, j, produce an output value x_j which is a function of its inputs, x_i, from the preceding layer(s) and the weight for the connection between the processing elements i to j, w_{ij}.

$$\begin{aligned} v_j &= \sum_{i=0}^{n} w_{ij} x_i \\ x_j &= f(v_j) = \frac{1}{1+e^{-v_j}} \end{aligned}$$

The output units are the same as for a Multi-Layer Perceptron network and the backpropagation algorithm by Rumelhart *et al,* [15]. Element input x_0 is the bias term fixed at +1.

Hidden layer elements behave the same as those in a Boltzmann network, [31], having stochastic behaviour,

$$x_j = \begin{cases} 1, & \text{probability } f(v_j) \\ 0, & \text{probability } 1 - f(v_j) \end{cases}$$

It should be noted that all the processing elements use a sigmoidal, squashing or logistic function. Output layer units use the function directly to form their output values whereas, for hidden layer units, the function generates the probability of the neuron producing a one or firing. The hidden layer processing elements have a stochastic behaviour. In this network expected activity does not propagate from hidden units in the way that deterministic activations in an MLP network do.

The performance of the network to produce the desired output must be assessed and the network trained to produce a better approximation to the desired output.

Denoting the actual network output as $\mathbf{Y}$,

$$\mathbf{Y} = (y_1, y_2, \ldots, y_N)$$

where the y_i are N output units, this is purely a renaming of the x_j values to y_j values for the output layer, and letting the desired network output be $\mathbf{D}$,

$$\mathbf{D} = (d_1, d_2, \ldots, d_N)$$

A performance measure can be defined as the mean square of the difference between desired and actual output.

$$\varepsilon = \frac{1}{N} \sum_{i=1}^{N} (d_i - y_i)^2 \tag{2.10}$$

$$0 \leq \varepsilon \leq 1$$

This performance measure or network error is used as the basis for improving the network response. Output layer processing elements and hidden layer processing elements have their weights updated differently.

Output layer processing elements again operate for updating as per Rumelhart *et al*, [15], in that the weights are updated by the backpropagation method, that is, a gradient descent occurs.

$$\Delta w_{ij} = \rho(d_j - y_j) f'(v_j) x_i$$

where $f'(v_j)$ is the derivative of the function $f(v_j)$,

$$f'(v_j) = f(v_j)(1 - f(v_j)) = y_j(1 - y_j)$$

ρ is a training gain term which affects how great the adjustment in the weight, w_{ij}, is made. As the hidden layer processing elements have a stochastic behaviour the error, ε, is random thus the adjustment in weights for the output layer will be random.

Hidden layer processing element weight update is accomplished by means of a broadcast reinforcement signal, r, which is sent to all hidden processing elements. This is simpler than backpropagating an error through previous processing elements from output towards the input. All weight updates can be performed simultaneously rather than waiting for other layers of elements to complete their updates as is the case for example in the back-propagation algorithm. Two schemes were proposed by Barto and Jordan [2], for weight updating in the hidden layers using the value of ε, the mean squared error between desired and actual network output, eq.(2.10). These schemes are the the P-model A_{R-P} and the S-model A_{R-P}. The P-model A_{R-P} is a binary reinforcement technique for hidden element weight updating, while the S-model A_{R-P} is a proportional reinforcement method for updating the hidden element weights.

P-model $\mathbf{A_{R-P}}$

The reinforcement signal r for updating the hidden layer weights has a probabilistic binary value depending upon ε,

$$r = \begin{cases} 1, & \text{probability } (1-\varepsilon) \\ 0, & \text{probability } \varepsilon \end{cases}$$

The better the network is at producing the desired output the greater the probability of a 1, implying success. Hidden processing elements have their weights updated according to the following rule,

$$\Delta w_{ij} = \begin{cases} \rho(x_j - f(v_j))x_i & \text{if } r = 1 \\ \lambda\rho(1 - x_j - f(v_j))x_i & \text{if } r = 0 \end{cases} \tag{2.11}$$

ρ is the training gain affecting how much weights are adjusted, while λ is the *degree of asymmetry* between the size of the weight change for $r = 1$, viewed as success, and $r = 0$, viewed as failure, $0 \leq \lambda \leq 1$. If $\lambda = 0$ then the weight update strategy is a Reward-Inaction, else for $\lambda > 0$ the strategy is a Reward-Punish.

The qualitative way this scheme works for hidden elements is that for success, $r = 1$, the weights, w_{ij}, alter so that the probability of the processing element producing the same response for the same input pattern increases. Thus in a similar situation the same actions will be more likely to be performed by the network. If $r = 0$ and the network fails the weight changes are such that the probability of the processing elements producing the same response for similar input patterns are reduced. The weight changes for failure are governed by λ so weight adjustment can also be scaled for failure of the network relative to success by the network. The reward and punish could be decoupled such that two separate gain terms are used, ie. ρ for reward and λ for punish, by removal of the ρ factor from the equation for the case $r = 0$ in eq.(2.11).

S-model $\mathbf{A_{R-P}}$

This scheme is simpler than the P-model with a real valued reinforcement signal, r, directly derived from the error, ε, as opposed to a probabilistic binary value for r.

$$r = 1 - \varepsilon$$

The better the network performance the smaller ε will be and the stronger the reinforcement, r. There is only the need for one weight updating algorithm,

$$\Delta w_{ij} = \rho(r(x_j - f(v_j)) + \lambda(1-r)(1 - x_j - f(v_j)))x_i \tag{2.12}$$

This scheme is simpler than the P-model A_{R-P}. It can be seen to reduce to the P-model A_{R-P} for values of $r = 0$ or $r = 1$.

2.8 Two New Extensions for Reinforcement Learning: Q-model and T-model $\mathbf{A_{R-P}}$

For the basic P-model and S-model A_{R-P} schemes tested by Barto and Jordan and reviewed in §2.7.1 two forms of weight adjustment are used in each method, namely the gradient descent at the output layer processing elements and the reinforcement at the hidden layer processing elements. To have just a single weight adjustment scheme would be better for hardware implementation purposes to keep the design as simple and uniform as possible. By eliminating the gradient descent at the output processing elements two new architectures may be evolved, the Q-model A_{R-P} and the T-model A_{R-P} based upon the P-model A_{R-P} and the S-model A_{R-P} respectively.

A second variation which was incorporated into the Q and T-model A_{R-P} was that all the neurons in the underlying network model now operate stochastically. The neurons have a binary output based on the sigmoid transform of the weighted sum of inputs. In the original form only the hidden layer neurons had a stochastic output while the output layer neurons operated deterministically. The weight values for the network are still real valued and continuous. This network model with associated learning strategy is now beginning to model the style of network that can be formed from the developed hardware stochastic neuron, and this is one of the reasons for investigating the reinforcement learning approach.

Q-model $\mathbf{A_{R-P}}$ The Q-model A_{R-P} is derived from the P-model A_{R-P} with all weights subjected to probabilistic binary reinforcement. The adaption strategy for all neuron weights in all layers are based on a reinforcement signal, r, which has a probabilistic binary value dependant upon the network error, ε.

$$\varepsilon = \frac{1}{N}\sum_{i=1}^{N}(d_i - y_i)^2$$

$$0 \leq \varepsilon \leq 1$$

d_i is the desired neuron output value and y_i is the actual neuron output value for N neurons.

Thus

$$r = \begin{cases} 1, & \text{probability } (1-\varepsilon) \\ 0, & \text{probability } \varepsilon \end{cases}$$

As a network produces an output closer to that desired, the greater the probability of a favourable reinforcement signal, $r = 1$. All processing elements now have their weights adjusted according to the following rule

$$\Delta w_{ij} = \begin{cases} \rho(x_j - f(v_j))x_i & \text{if } r = 1 \\ \lambda\rho(1 - x_j - f(v_j))x_i & \text{if } r = 0 \end{cases}$$

ρ is the training gain affecting how much weights are adjusted, while λ is the *degree of asymmetry* between the size of the weight change for $r = 1$, viewed as success, and $r = 0$, viewed as failure, $0 \leq \lambda \leq 1$. If $\lambda = 0$ then the weight update strategy is a Reward-Inaction, else for $\lambda > 0$ the strategy is a Reward-Punish.

T-model $\mathbf{A_{R-P}}$ The T-model A_{R-P} is similarly derived from the S-model A_{R-P} adaption strategy. All weights are now varied due to a real valued reinforcement signal, r, derived directly from the error, ε.

$$r = 1 - \varepsilon$$

As the network produces an output closer to that desired the greater the value of the reinforcement signal. All processing elements in this model have their weights adjusted according to the following rule

$$\Delta w_{ij} = \rho(r(x_j - f(v_j)) + \lambda(1 - r)(1 - x_j - f(v_j)))x_i$$

2.8.1 Evaluating the Four $\mathbf{A_{R-P}}$ Strategies

In §2.3.3 simple feedforward networks were used to assess the capabilities and learning rates of an MLP with the backpropagation algorithm. As a comparison the two styles of problem which had been used with the MLP evaluation were repeated for the four A_{R-P}

algorithm variants, namely the encoder/decoder problem and **XOR** problem.

The encoder/decoder problem was the same style of 8–3–8 network of artificial neurons while the **XOR** used a 2–2–1 architecture of artificial devices. After difficulty was experienced gaining favourable results for the Q and T-model algorithm simulations with the 8–3–8 problem, but success was achieved with the 2–2–1 **XOR**, a new set of simulations for a reduced 4–2–4 encoder/decoder network were performed for the Q and T-model. A spread of training parameters were used with varying training gain ρ and asymmetry λ for each of the learning models.

A simulation run consisted of presenting a pattern to the network and noting the network's response. The weights of each artificial neuron were updated and a new pattern selected at random from the input set and presented. After a given number of pattern presentations the network performance was calculated by presenting each of the input patterns in turn and determining the RMS error value. The average of the RMS error value is taken as a measure of the overall performance of the network.

P-model and S-model $\mathbf{A_{R-P}}$ For both the P and S schemes rapid initial descent governed by the value of ρ the training gain is observed. In general the greater the value of ρ the faster the rate of descent but with diminishing returns, Figure 2.34 and Figure 2.35. In both these cases the long term adaption levels out to an offset value greater than zero as illustrated in Figure 2.36. NB. For all of these simulation runs $\lambda = 0$.

By addition of a degree of asymmetry, $\lambda > 0$, both the P and S models are able to produce an improved adaption result as illustrated by Figure 2.37. Even a very small value of λ is significant in improving the adaption capabilities, Figure 2.38. If, however, the value of λ is too large, then the P and S algorithms fail to adapt to an optimum solution but as with the case of $\lambda = 0$ tend to a non-zero value. The error oscillates more vigorously about this offset level though.

Using the P and S-model algorithms to train a 2–2–1 network of neurons to perform the non-linearly separable problem of the **XOR** proved as difficult as with more sophisticated algorithms. For any of the combinations of ρ and λ attempted the network could not be trained to the appropriate value with either algorithm. It was found that a very small degree of asymmetry was required and that the number of pattern presentations made to the network was extremely large for the network error to tend to zero, Figure 2.39 and Figure 2.40. For this case there is a rapid initial descent as with the 8–3–8 encoder/decoder but the improvement to remove the last portion of error is very slow.

Q-model and T-model $\mathbf{A_{R-P}}$ As with the previous two P and S variants the 8–3–8 encoding problem was tackled with these new Q and T-model versions. As each output neuron now produces an integer response 1 or 0 the performance measure, RMS Error, will now be in discrete levels. The trend of increasing the gain ρ to increase the rate of learning could not be observed in the performance plots. Varying the amount of asymmetry did not aid in the adaption process for either the Q or T-models, unlike the P and S-models, the network performance was poor and varied widely even with small value eg. $\lambda = 0.005$ as exemplified by Figure 2.41.

Surprisingly, when the Q and T-models are applied to the **XOR** problem with a small non-zero value for λ the problem could be adapted to, Figure 2.42 and Figure 2.43. Note the highly quantitised performance measure for the network and learning algorithms which provide a possible insight into the problem of adaption with the 8–3–8 encoder/decoder.

With the output of each neuron being either correct or incorrect with respect to the probability given by a function of its weighted inputs the opportunity for the network to obtain a strong reinforcement signal, ie. the probability that all outputs are correct, to enhance its performance is limited. The training time necessary may therefore be longer than that allocated for the above experiments.

Returning now to the encoder/decoder style configuration but with a reduced size of problem, ie. 4–2–4, it can be seen from Figure 2.44 and Figure 2.45 that the network with either the Q or T-model reinforcement training algorithm can now work in the time allocated. The assortment of values for gain and asymmetry presented are due to the fact that conversion to a satisfactory result is not always possible. Given one set of gain and asymmetry values the algorithm may not converge, but given new initial random weight values the network may converge. It can be seen in Figure 2.44 that for $\rho = 0.9$ and $\lambda = 0.03$ the system is probably stuck in a local minimum before being able to escape at around 50000 presentations.

2.9 Conclusions

In this chapter the aim has been to provide a critical review of four key neural network architectures, the MLP §2.3.2, the Kohonen Self-Organising Feature Map §2.4, the Hopfield Net §2.5 and Boltzmann Machine §2.6 in order to determine the most appropriate attributes for hardware implementation and on-line learning. The first two networks were simulated in software in order to gain a fuller appreciation and understanding of their functionality.

Several architectures and paradigms utilising reinforcement learning techniques have been reviewed §2.7. These algorithms are of particular interest since they usually use the minimum amount of information which has to be fed back through the network. The two learning models, P and S, presented by Barto *et al* have been demonstrated to function as specified. The two systems were found to rely on a small punishment signal in order to gain their best performance.

Building on these two models their respective reinforcement strategies were extended to the output layer of a network. In addition, the output layer neurons were configured such that their output was probabilistic as per the hidden layer. It was found that feeding a single reward or punishment signal to every neuron, it was possible to train the network to perform the two demonstration tasks of the 4–2–4 encoder/decoder and the 2–2–1 **XOR** problem. Again it was found that the asymmetry term, λ, was important in the network adaption performance. When the larger 8–3–8 encoder/decoder problem was attempted with these new learning algorithms they did not converge in the time used to train them, there may thus be a scalability issue which needs to be addressed in using these methods.

It can be seen that there are many and varied algorithms used in the study of ANNs. The research into these algorithms is normally conducted in software models. It has been highlighted throughout that NNs are essentially a parallel processing technique consisting of many simple processing elements which are interconnected. The hardware design of the processing elements is thus a key issue if the most benefit is to be gained from these systems. The following chapter, §3, provides a review of possible hardware techniques which may be used to form ANNs. Included in this review are several commercially available devices.

The method of stochastic pulse rate encoded signals is discussed in the hardware review, it is pursued further by an explanation of the coding techniques and processing circuits in §4. This suite of circuits is extended with novel circuit designs relevant to ANNs before an actual hardware neuron design is discussed, developed, tested and operated in the following chapters of this thesis.

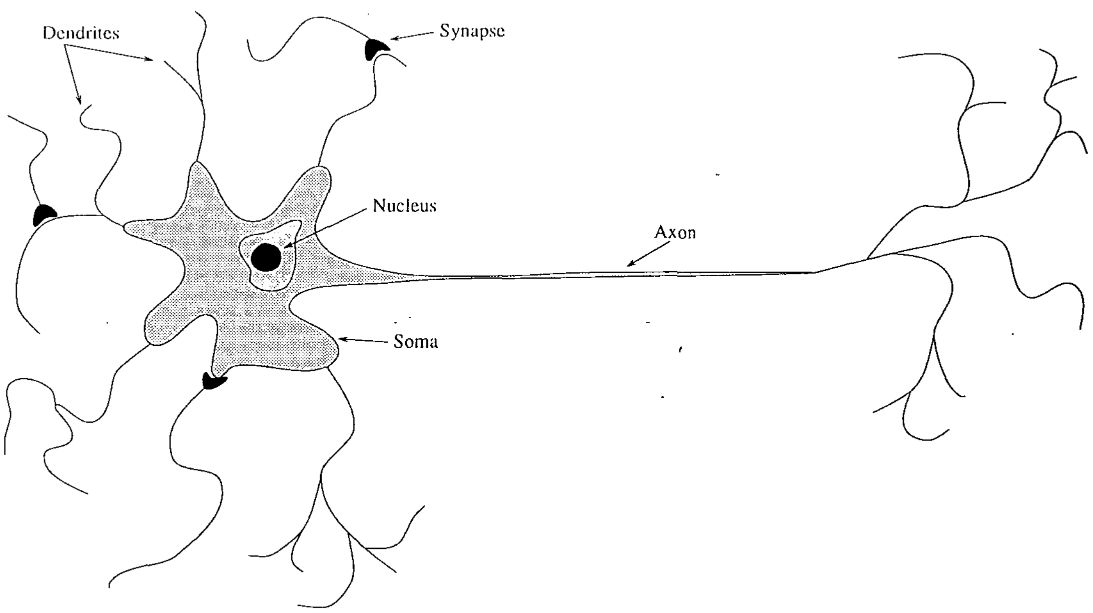


Figure 2.1: Illustration of a Biological Neuron Structure. *Artificial neurons model a simplified structure of a biological neuron. A single, simple, processing element with many inputs and one output.*

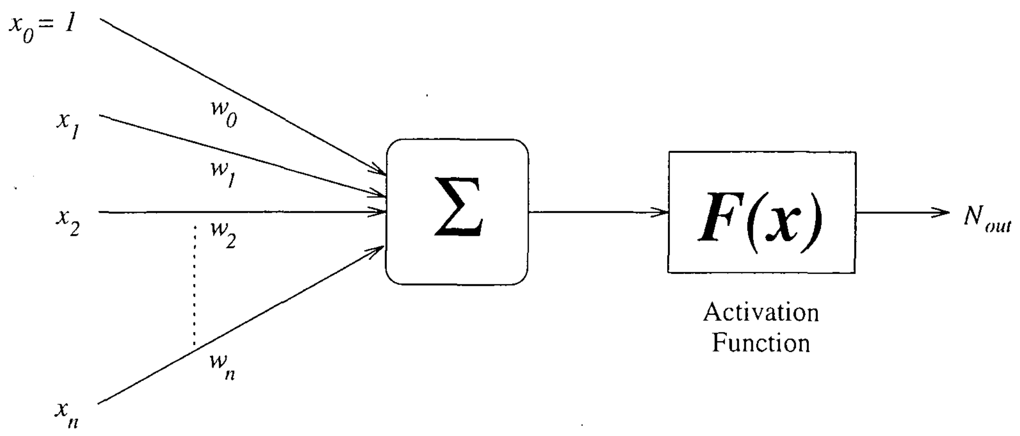


Figure 2.2: General Artificial Neuron Architecture of McCulloch and Pitts. *This consists of weighted input values which are summated and then passed through an activation function.*

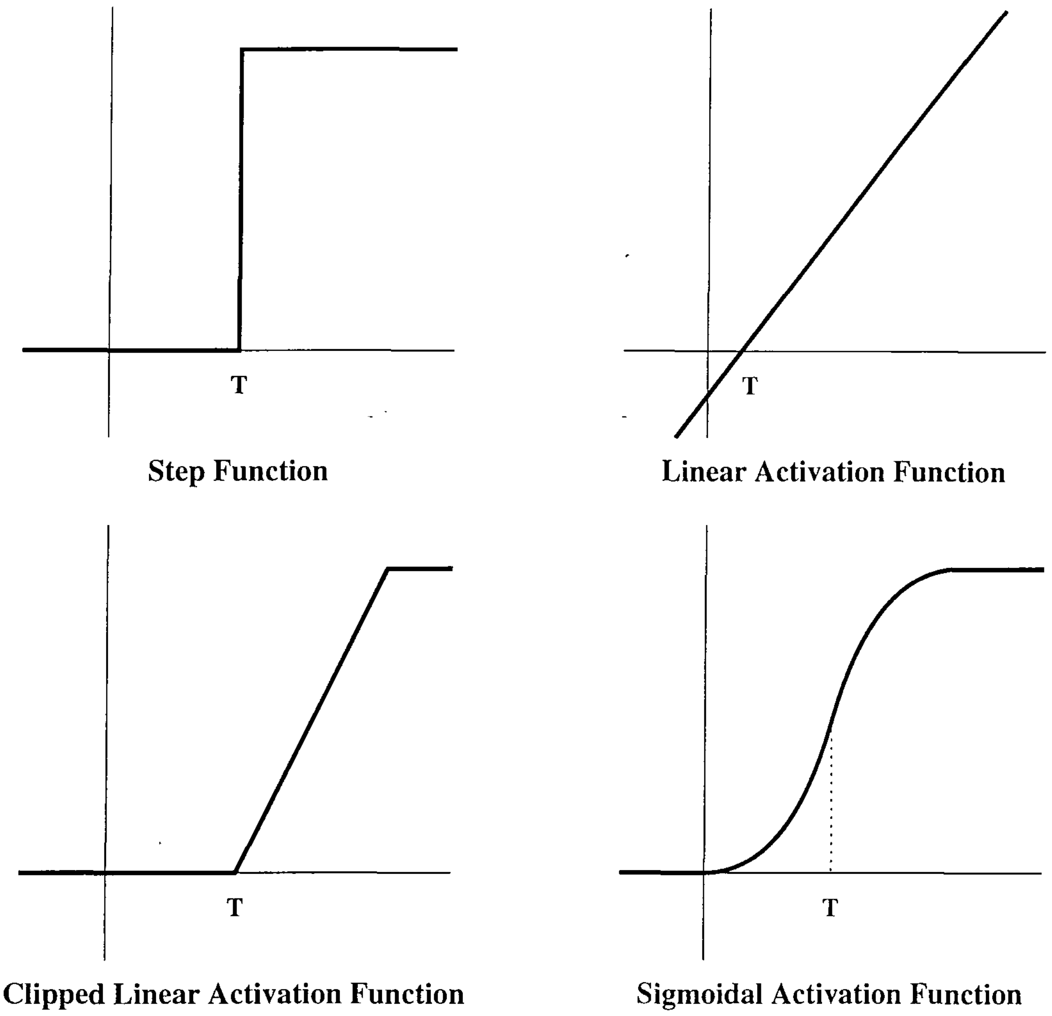


Figure 2.3: Common Neuron Activation Functions. *The Step Threshold function was the original proposed by McCulloch and Pitt. Alternative activation functions are illustrated, all but the Linear Activation Function constrain the output range of the neuron.*

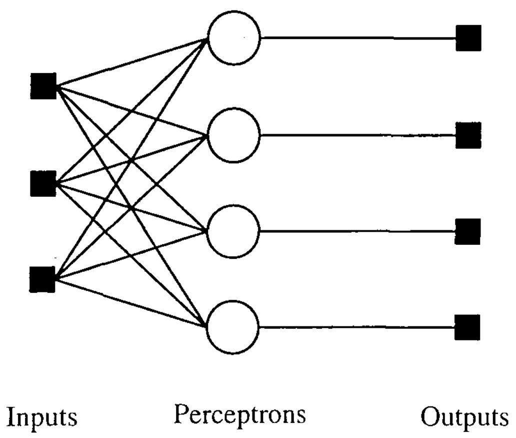
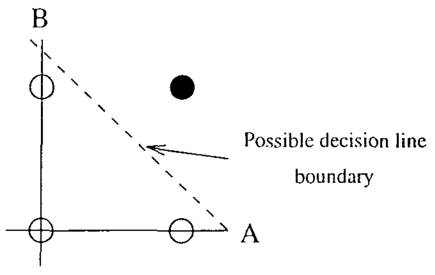


Figure 2.4: Single layer perceptron configuration. *There is only a single processing layer in this structure with no feedback connections and no connections across the network from one perceptron to another.*

AND Function

A	B	Response
0	0	0
0	1	0
1	0	0
1	1	1



XOR Function

A	B	Response
0	0	0
0	1	1
1	0	1
1	1	0

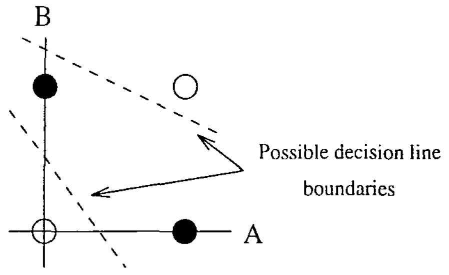


Figure 2.5: Example of AND and XOR functions for the Perceptron. *The AND function is linearly separable, a single decision line can divide the two output domains. The XOR function is not linearly separable, more than one decision line is necessary to divide the two output domains.*

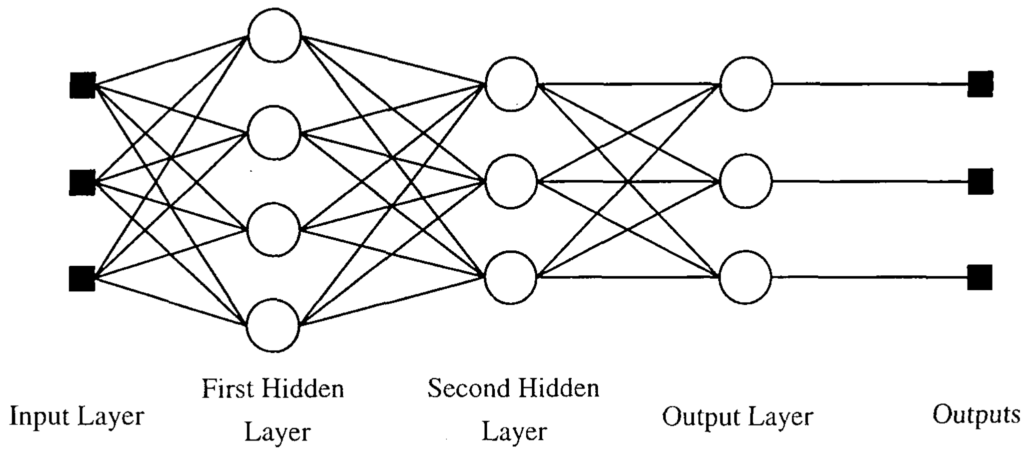


Figure 2.6: Three layer fully connected MLP configuration. *As for the SLP, all connections are feedforward to the next layer only with no connections between neurons in the same layer.*

```
layers              3
neurons per layer   8  3  8
training gain       0.5
training momentum   0.2
tv                  8
training type       r
inspect rate         10
training group size 1
epochs               2000
ip 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
op 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
ip 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
op 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
ip 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0
op 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0
ip 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
op 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
ip 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
op 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
ip 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
op 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
ip 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
op 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
ip 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
op 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

Figure 2.7: Example of the file `setup.mlp`. *This file is used to configure the basic MLP simulator written to demonstrate and verify the operation of MLPs.*

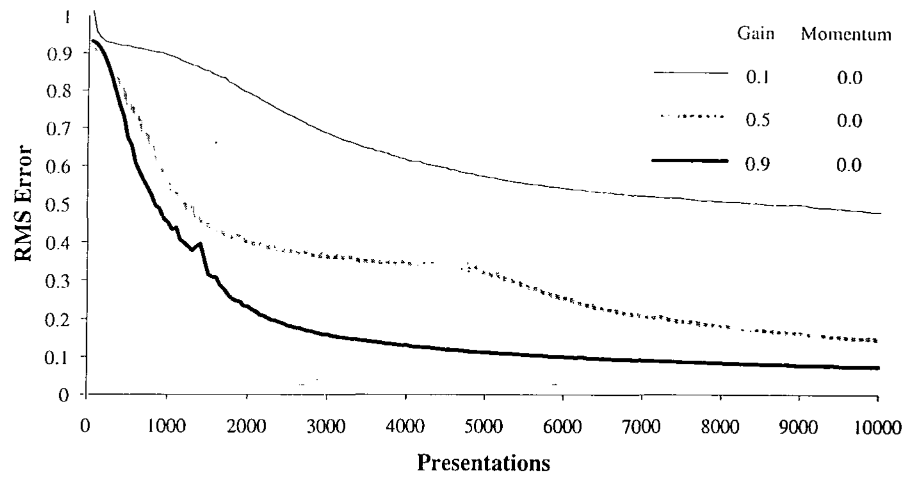


Figure 2.8: Error curves for 8–3–8 coder/decoder MLP, Random presentation. *Increasing the gain term for backpropagation increases the rate of reduction in RMS Error.*

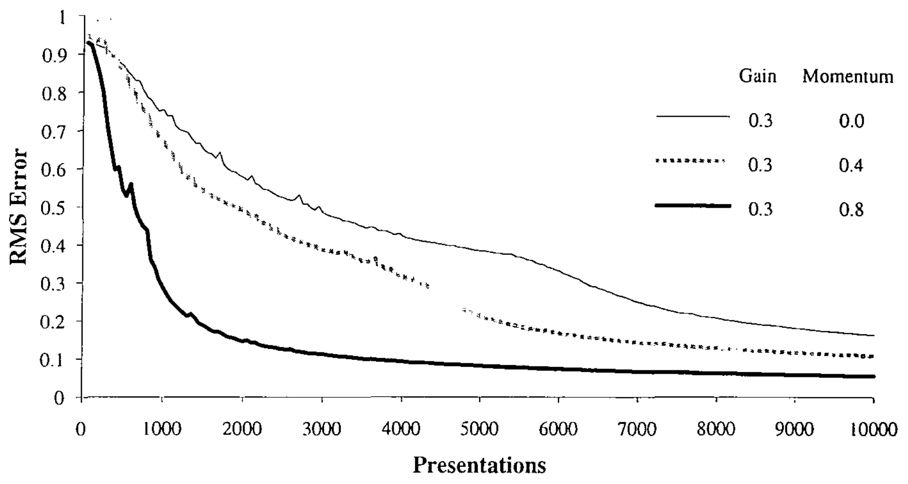


Figure 2.9: Error curves for 8–3–8 coder/decoder MLP, Random presentation. *Increasing the momentum term for backpropagation increases the rate of reduction in RMS Error.*

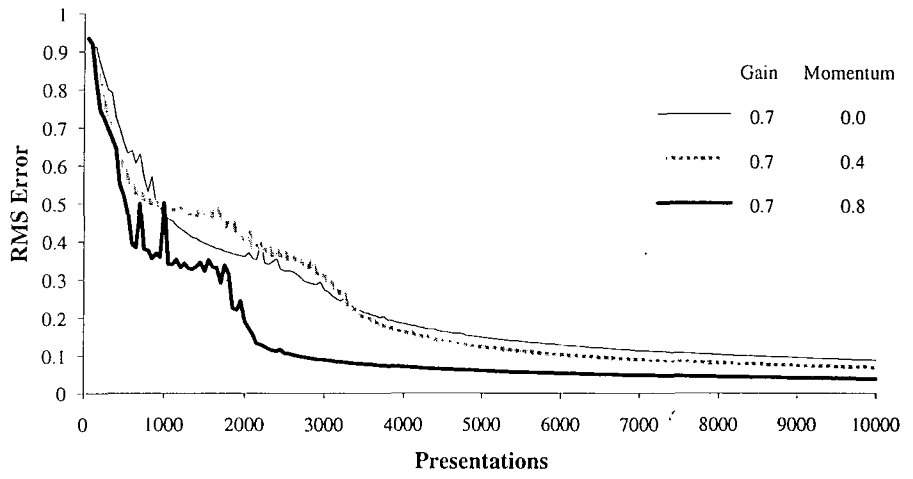


Figure 2.10: Error curves for 8–3–8 coder/decoder MLP, Random presentation. *Increasing the momentum term for backpropagation increases the rate of reduction in RMS Error, but for large values of gain and momentum the decrease in RMS Error is noisier and the convergence point is noisier, this is not obvious from these though.*

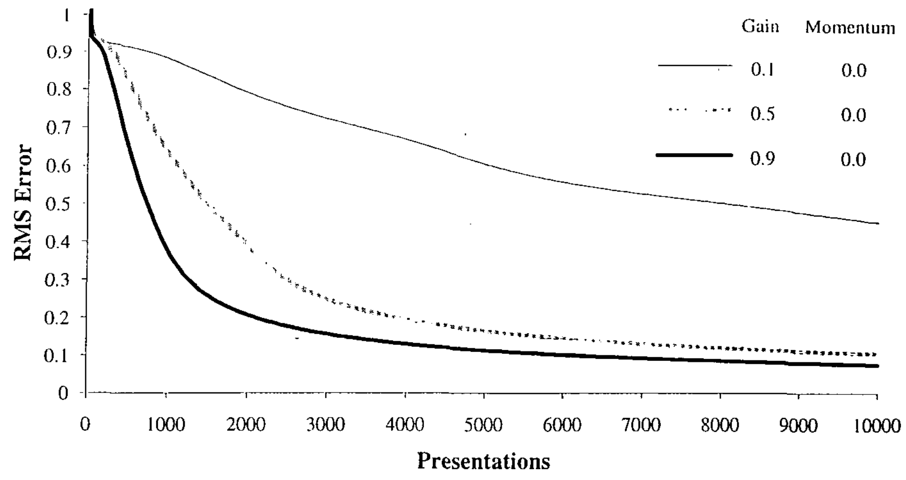


Figure 2.11: Error curves for 8–3–8 coder/decoder MLP, Batch presentation. *Increasing the gain term for backpropagation increases the rate of reduction in RMS Error.*

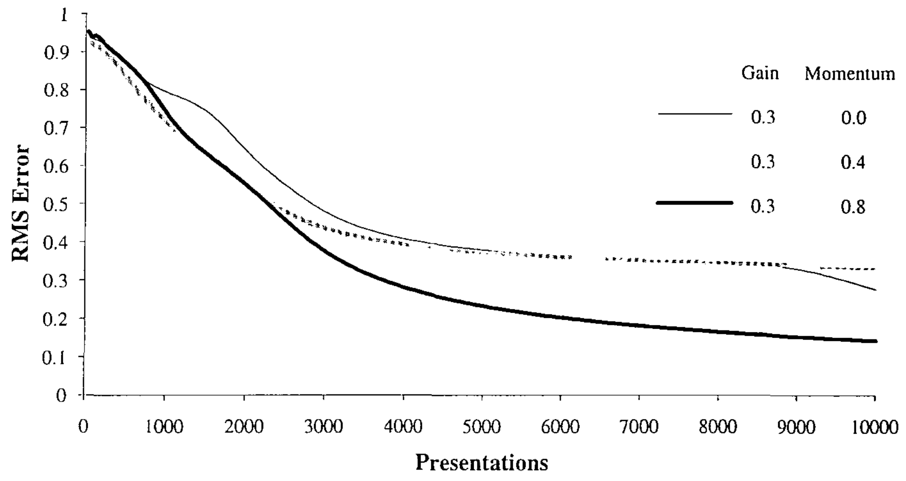


Figure 2.12: Error curves for 8–3–8 coder/decoder MLP, Batch presentation. *Increasing the momentum term for backpropagation increases the rate of reduction in RMS Error.*

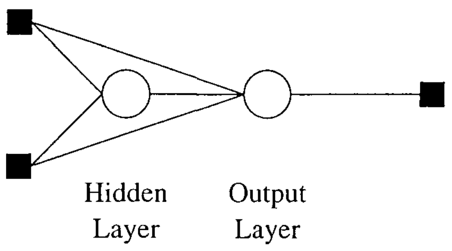


Figure 2.13: Rumelhart *et al* network architecture to solve the XOR problem. *Simplified network for solving the XOR problem. Note, however, that feedforward connections from the input layer pass directly to the output layer.*

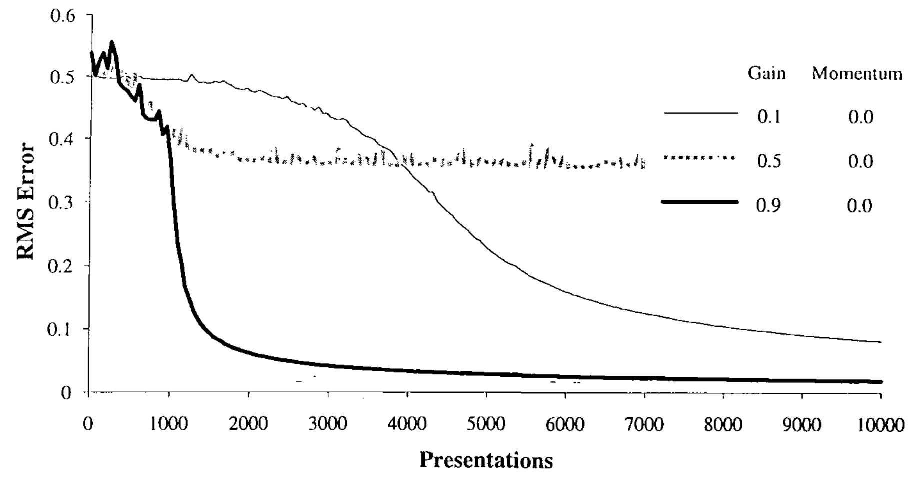


Figure 2.14: Error curves for 2–2–1 XOR MLP. *In general, increasing the gain term for backpropagation increases the rate of reduction in RMS Error. Note, a system will not always converge, eg.* $\eta = 0.5$, $\alpha = 0.0$

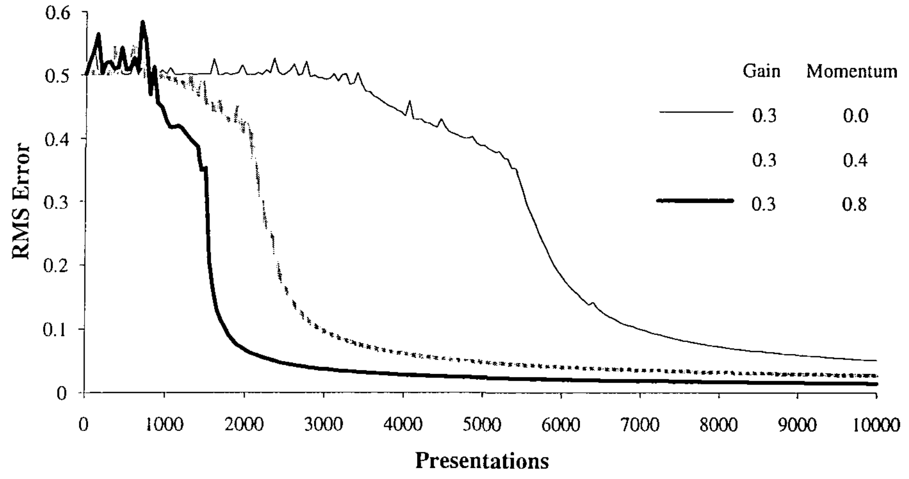


Figure 2.15: Error curves for 2–2–1 XOR MLP. *Increasing the momentum term for backpropagation increases the rate of reduction in RMS Error.*

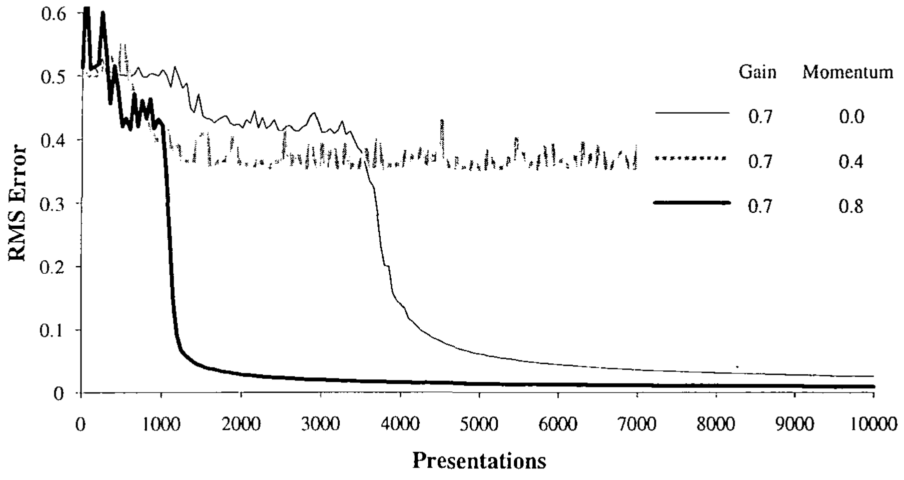


Figure 2.16: Error curves for 2–2–1 XOR MLP. *In general, increasing the momentum term for backpropagation increases the rate of reduction in RMS Error. Note a system will not always converge, eg.* $\eta = 0.7$, $\alpha = 0.4$

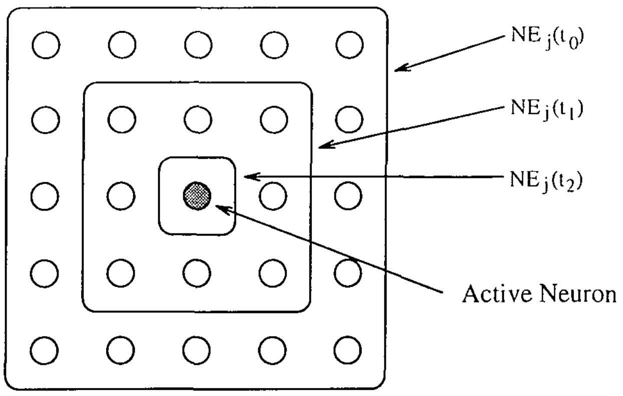


Figure 2.17: Kohonen Self-Organising Feature Map Network Neighbourhood Layout, 1. *Each neuron has eight nearest neighbours and the neighbourhood scales as 1–8–16.*

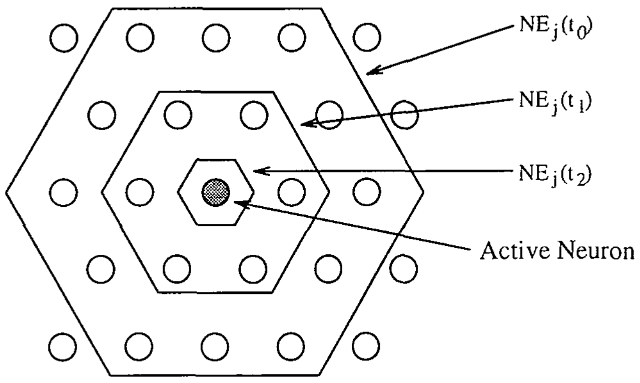


Figure 2.18: Kohonen Self-Organising Feature Map Network Neighbourhood Layout, 2. *Each neuron has six nearest neighbours and the neighbourhood scales as 1–6–12.*

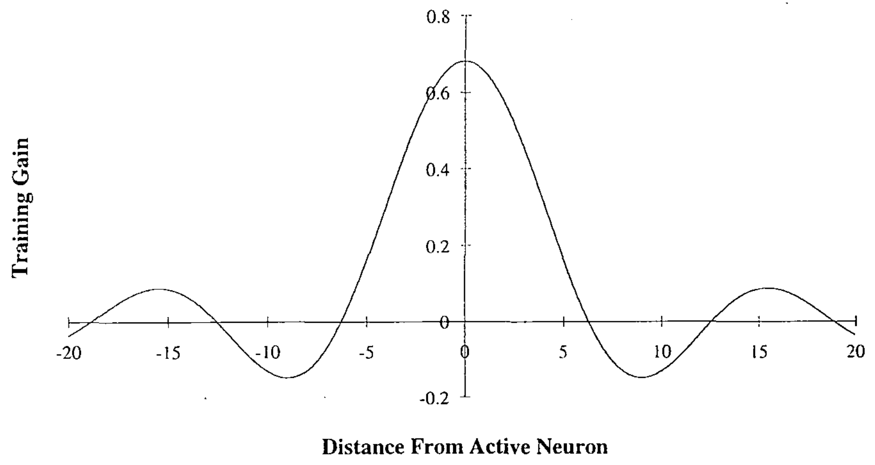


Figure 2.19: Variation in Training Gain, η, vs Distance from Active Neuron. *The influence of gain and neighbourhood size are combined within this single distribution. Negative values for gain as generated by this 'Mexican Hat' curve have proved successful in training Kohonen Self-Organising feature maps.*

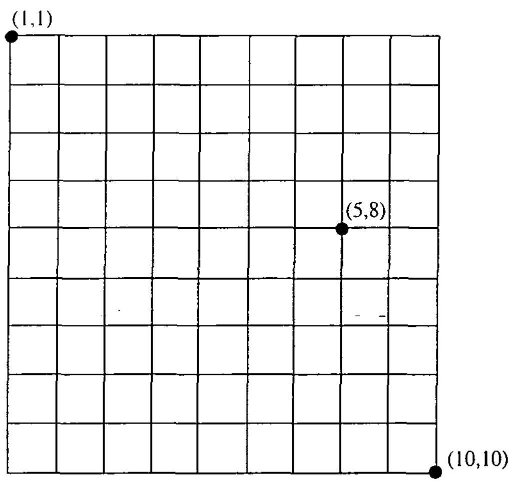


Figure 2.20: Ideal Uniform 10 by 10 Mesh. *A two dimensional array of 10 x 10 elements can be arranged as a uniformly spaced regular grid.*

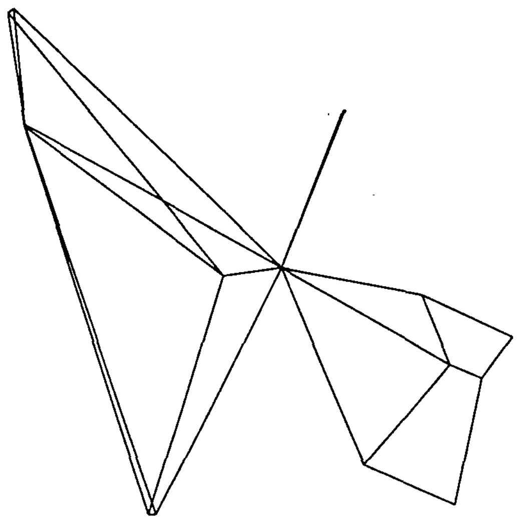


Figure 2.21: Kohonen Self-Organising Layer, 10 iterations. *After only a few iterations of the training algorithm the majority of the neuron responses are still concentrated around the central value. A large value of η and neighbourhood will be used to disperse the neuron responses throughout the output domain.*

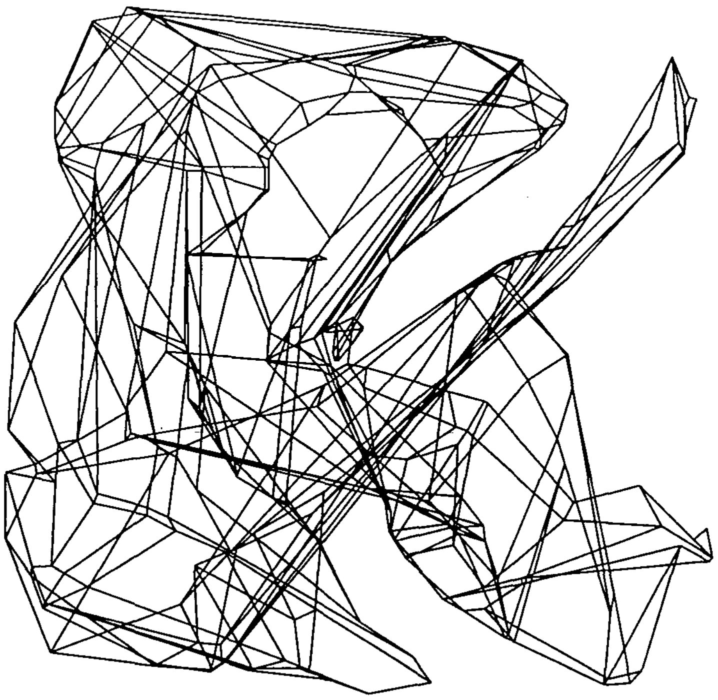


Figure 2.22: Kohonen Self-Organising Layer, 1000 iterations, Uniform (x,y) distribution. *The neuron responses have been distributed throughout the output domain. A 'twist' in the output map appears to exist. Provided there is enough energy within the system ie. large η and neighbourhood, the training algorithm should unravel this twist.*

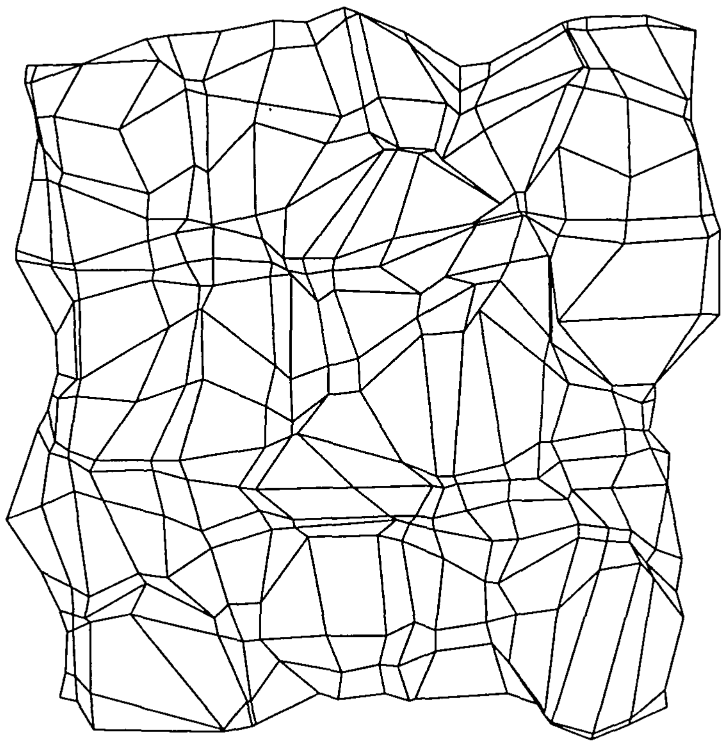


Figure 2.23: Kohonen Self-Organising Layer, 100000 iterations, Uniform (x,y) distribution. *The basic structure of the regular grid has been formed. The twist in the response has been undone.*

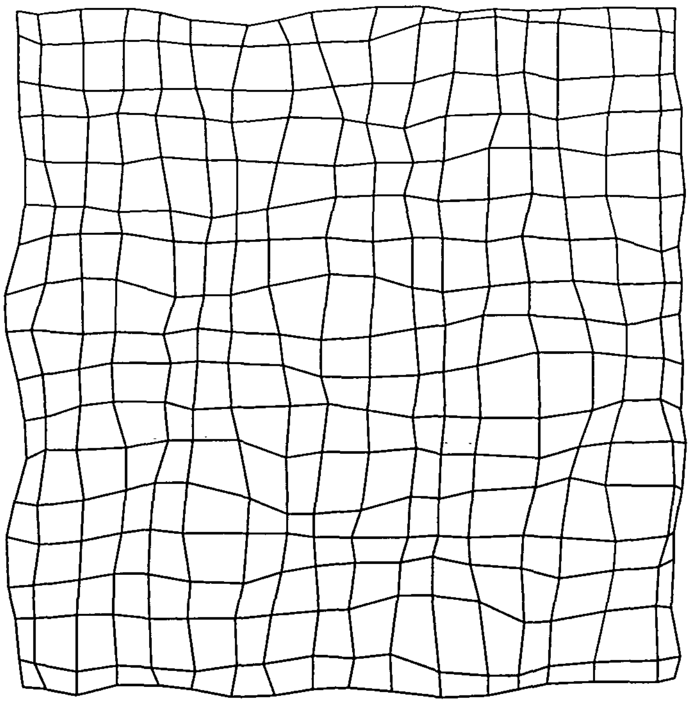


Figure 2.24: Kohonen Self-Organising Layer, 300000 iterations, Uniform (x,y) distribution. *The output grid has stabilised to the expected uniform structure for the uniformly distributed two dimensional inputs. Small values of η and neighbourhood will be used to continue fine tuning the network response.*

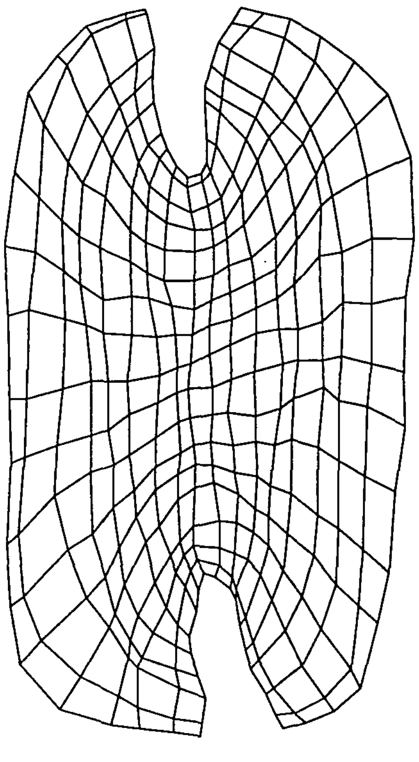
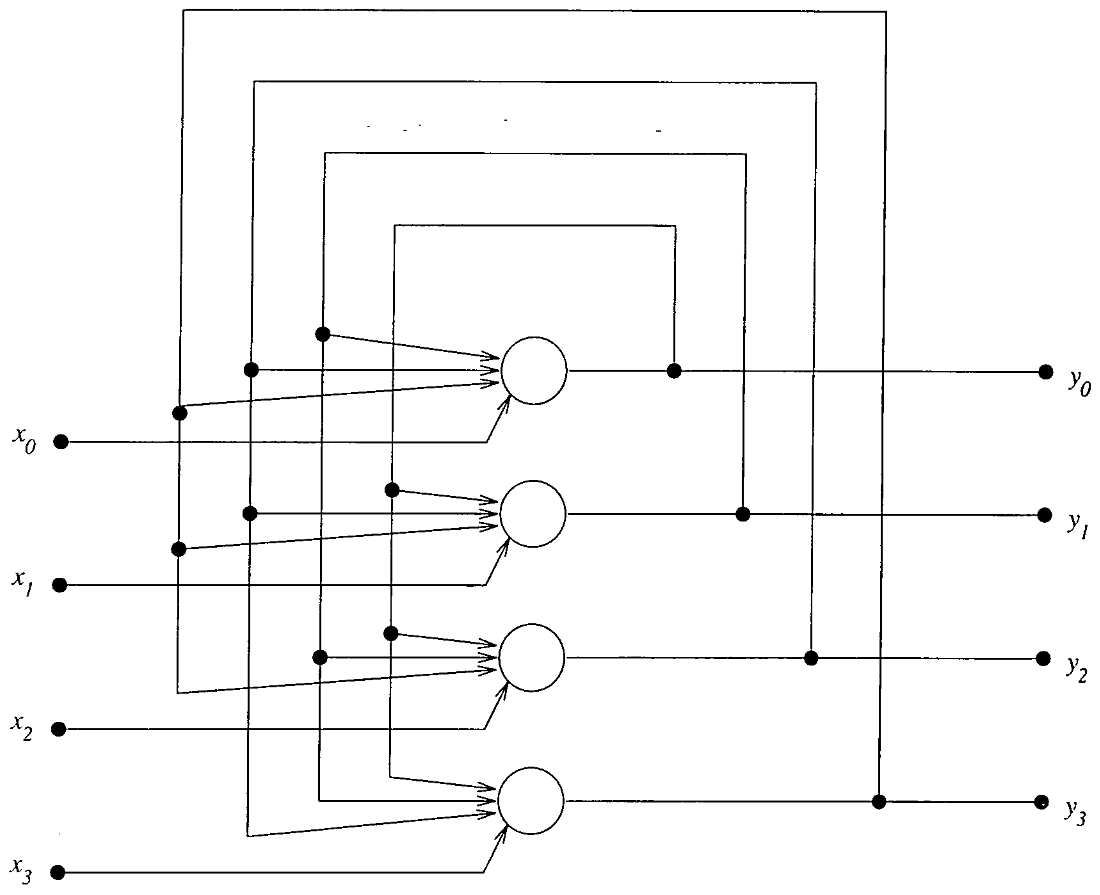


Figure 2.25: Kohonen Self-Organising Layer, 300000 iterations, Normal (x), Uniform (y) distribution. *With a concentration of information about the central value for the x-dimension the output map is pulled into a form where more neurons are used for areas where most information is present.*



x_0 - x_3 initially loaded input pattern

y_0 - y_3 output pattern which will be atable after convergence

Figure 2.26: General Architecture of a Hopfield Net, four neurons. *A Hopfield Net consists of a single layer of neurons with the feedback of their output to every neuron except themselves.*

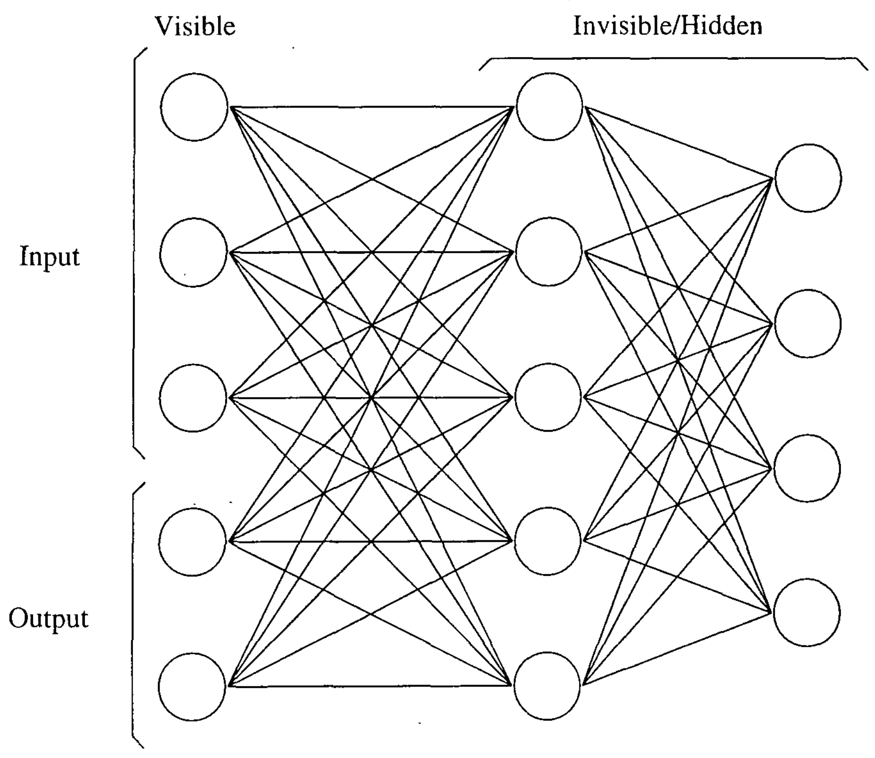


Figure 2.27: General Architecture of a Boltzmann Machine. *Neurons generate a stochastic output and can be divided into two classes, Visible and Invisible. Only visible neurons are connected to the outside and these can be further divided into Input and Output sub-classes.*

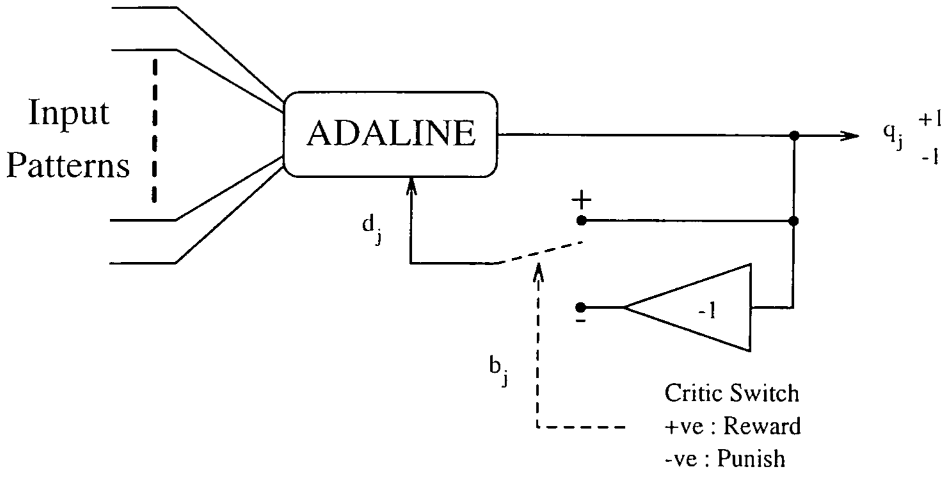


Figure 2.28: Criticised ADALINE. *Learning with a critic architecture, only a single +1, Reward, or -1, Punish, signal is used to update neuron weights.*

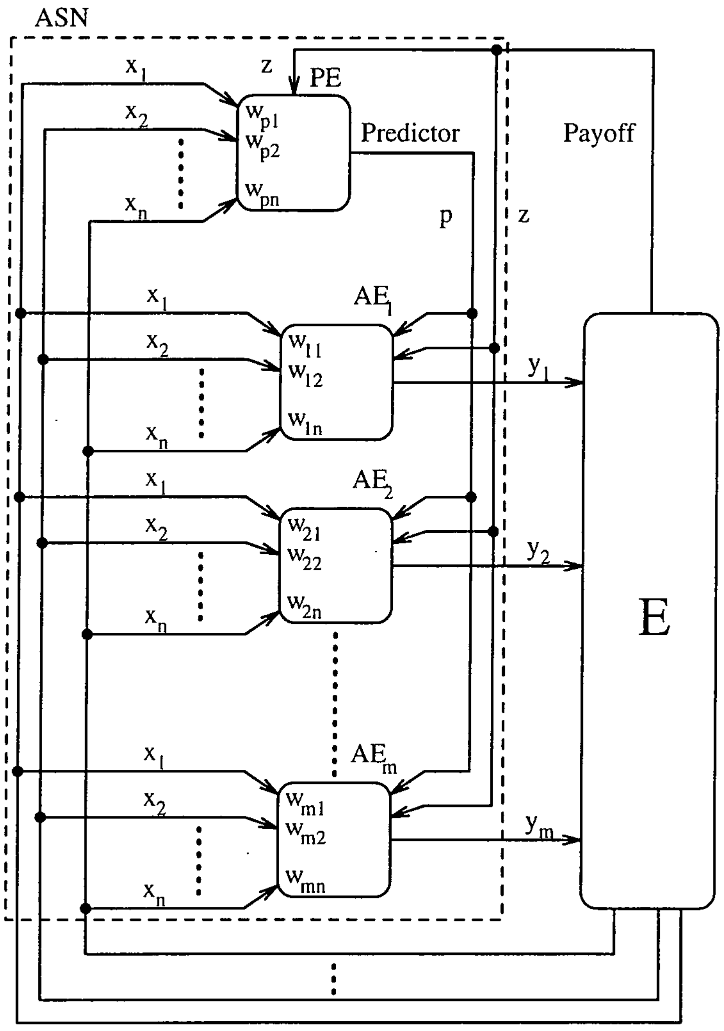


Figure 2.29: Associative Search Network Architecture. *The ASN has two types of processing elements, many Adaptive Elements, AE, and a single Predictor Element, PE. All processing elements are connected to the environment, E.*

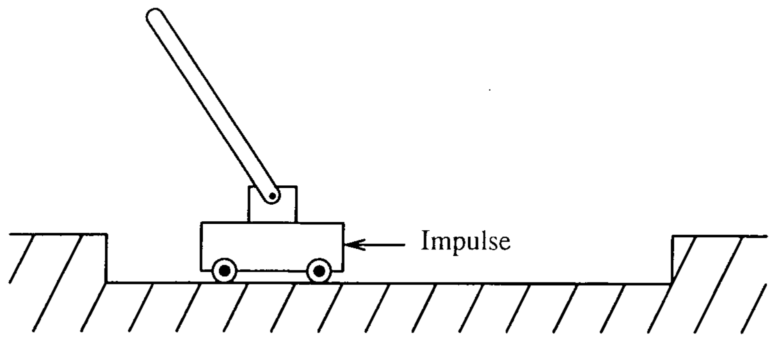


Figure 2.30: Cart-Pole balancing system. *By moving the cart appropriately the aim is to keep the pole in an upright position.*

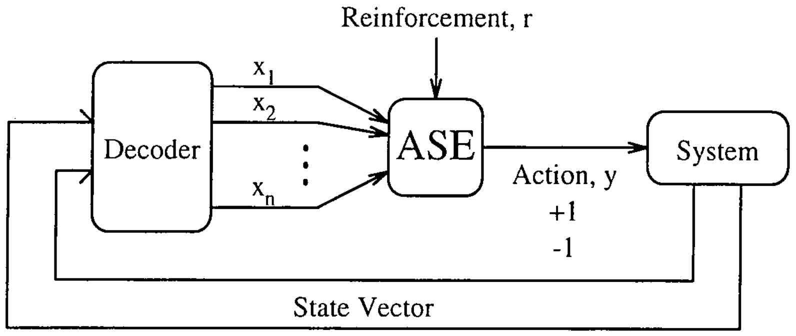


Figure 2.31: Associative Search Element (ASE) configuration. *The system environment status is decoded before feeding into the ASE. The reinforcement signal is only set at times of system failure. The system responses and rate of adaption were found to be poor.*

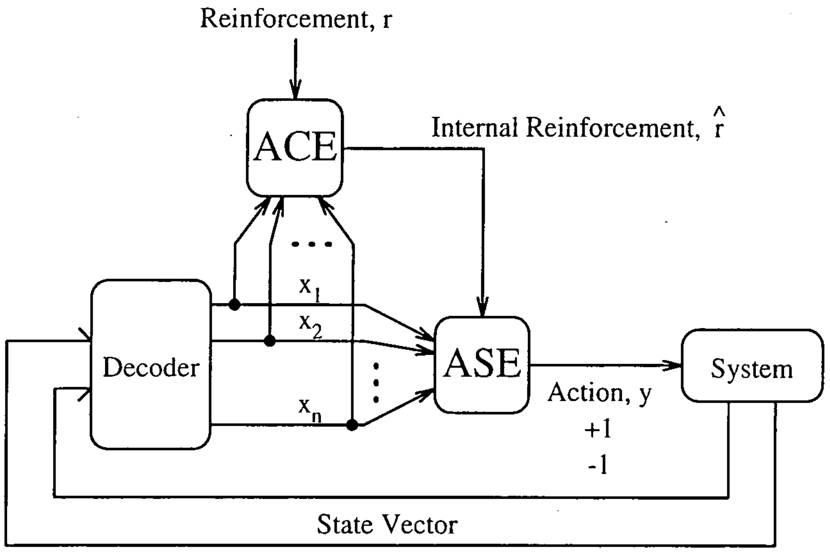


Figure 2.32: Associative Search Element with Adaptive Critic Element (ACE) Configuration. *Basic performance of the ASE system Figure 2.31, is enhanced by the inclusion of the ACE which generates a continuous value of internal reinforcement signal for every set of decoded outputs and reinforcement inputs.*

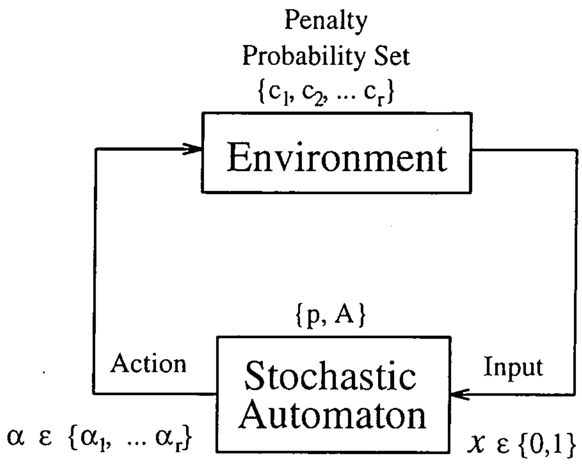


Figure 2.33: Learning Automaton.

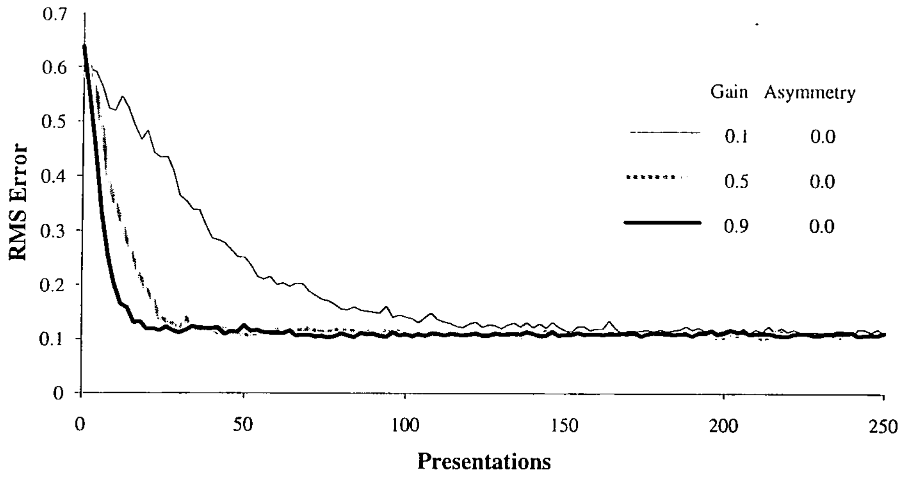


Figure 2.34: Initial adaption rate for 8–3–8 encoder/decoder P-model A_{R-P}. *It will be noted that increasing the gain, ρ, produces an increase in learning rate. There exists a constant error which the training algorithm can not overcome.*

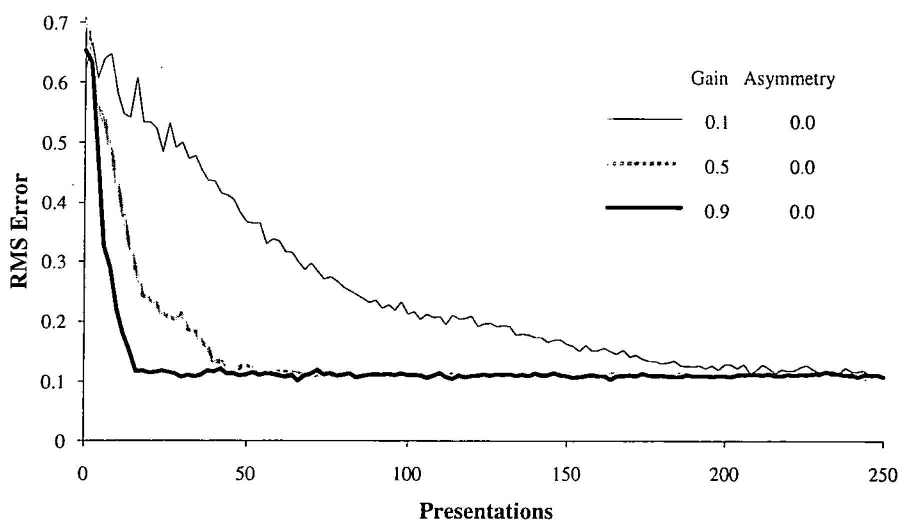


Figure 2.35: Initial adaption rate for 8–3–8 encoder/decoder S-model A_{R-P}. *It will be noted that increasing the gain, ρ, produces an increase in learning rate. There exists a constant error which the training algorithm can not overcome. By comparison with Figure 2.34 the P-model A_{R-P} is marginally faster at error reduction.*

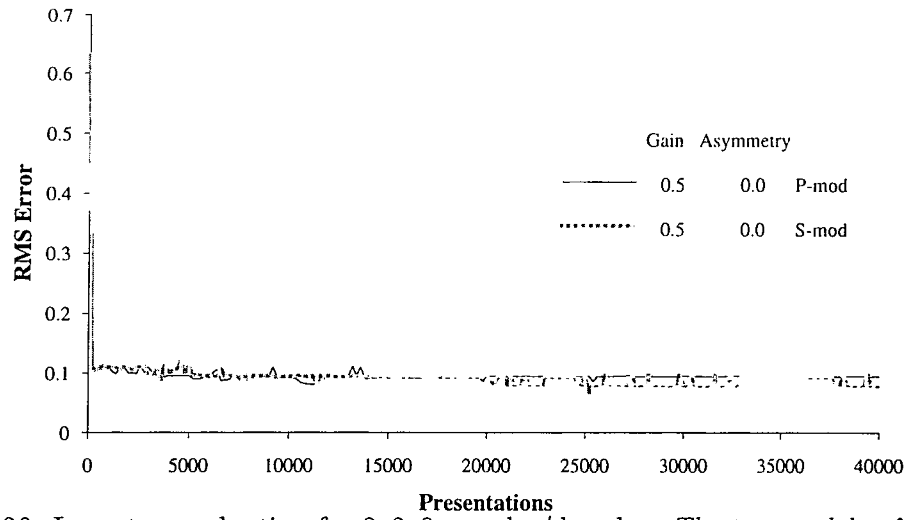


Figure 2.36: Long term adaption for 8–3–8 encoder/decoder. *The two models of network of Figure 2.34 and Figure 2.35 have be trained for a long period of time but remain with the same amount of network error.*

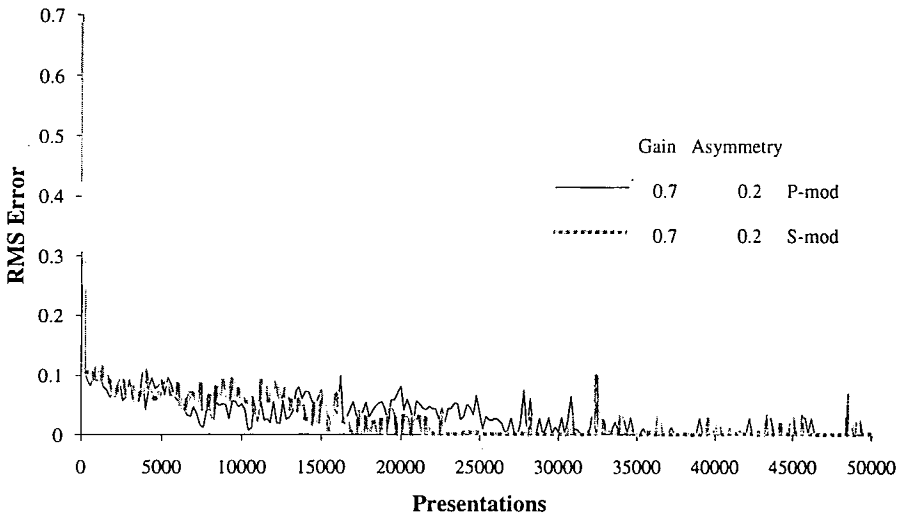


Figure 2.37: Long term adaption for 8–3–8 encoder/decoder with $\lambda > 0$. *It can be seen that increasing the value of asymmetry from zero aids the training of the network by reinforcement learning.*

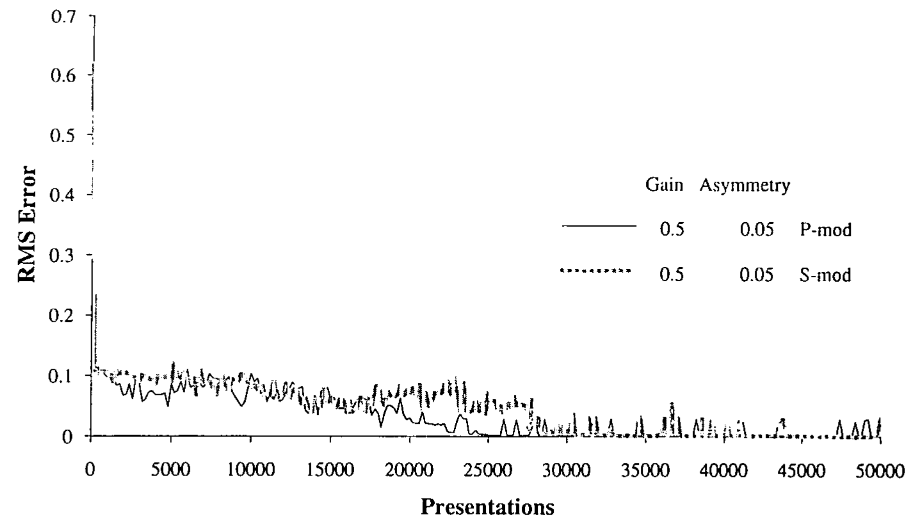


Figure 2.38: Long term adaption for 8–3–8 encoder/decoder with small λ. *By comparison with the previous Figure 2.34 and Figure 2.35 it can be seen that even a small degree of asymmetry is beneficial.*

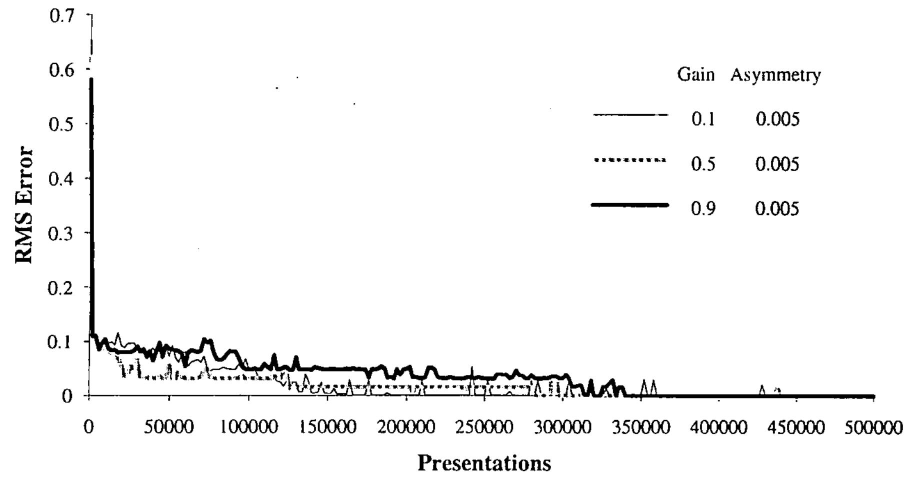


Figure 2.39: **XOR** learning P-model. *For adaption to occur such that the network error tends to zero it is necessary to use a very small value of λ and a long training period.*

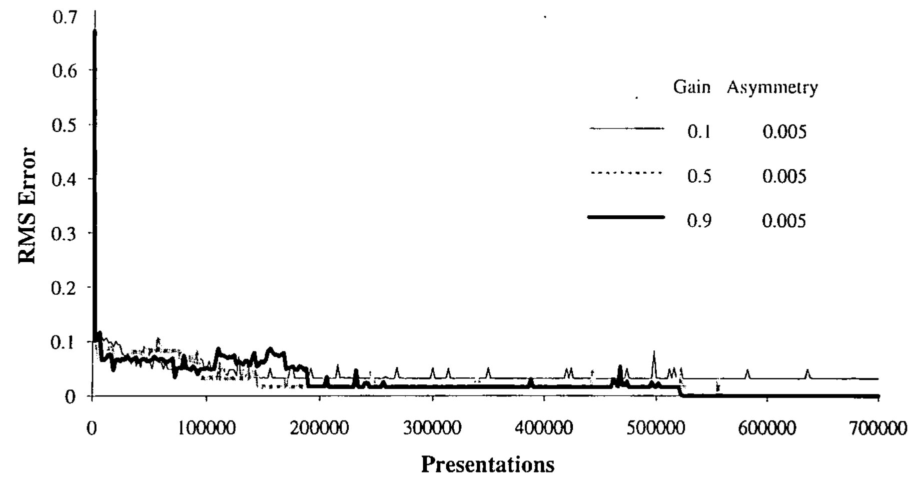


Figure 2.40: **XOR** learning S-model. *As per the P-model, Figure 2.39, a small value of λ was found to be necessary combined with a long training period for network error to tend to zero.*

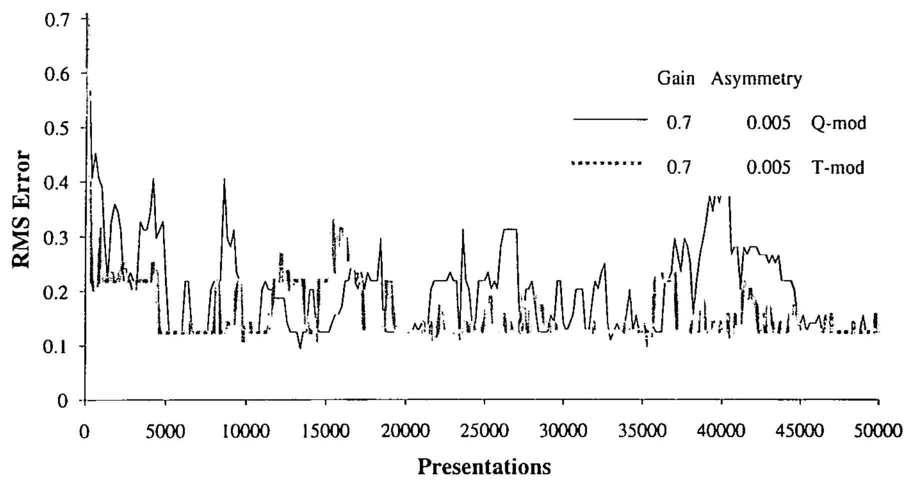


Figure 2.41: Poor learning of 8–3–8 by Q and T models. *This is an example of the poor adaption of the new A_{R-P} models and the inability to reduce the network error to zero even for small degrees of asymmetry.*

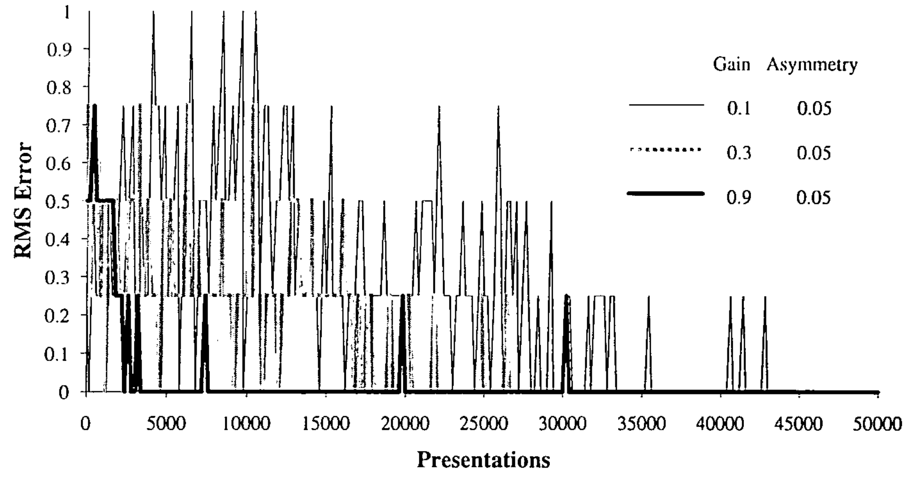


Figure 2.42: Q-model **XOR**. *Note that by comparison with the P-model results of Figure 2.39 the rate of adaption and learning is of the order of ten times faster. A highly quantised response is evident.*

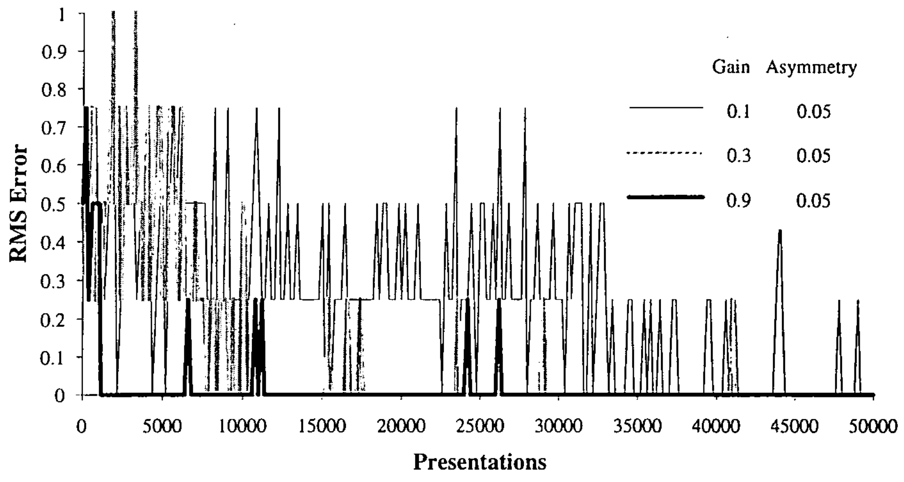


Figure 2.43: T-model **XOR**. *Note that by comparison with the P-model results of Figure 2.40 the rate of adaption and learning is of the order of ten times faster. A highly quantised response is evident.*

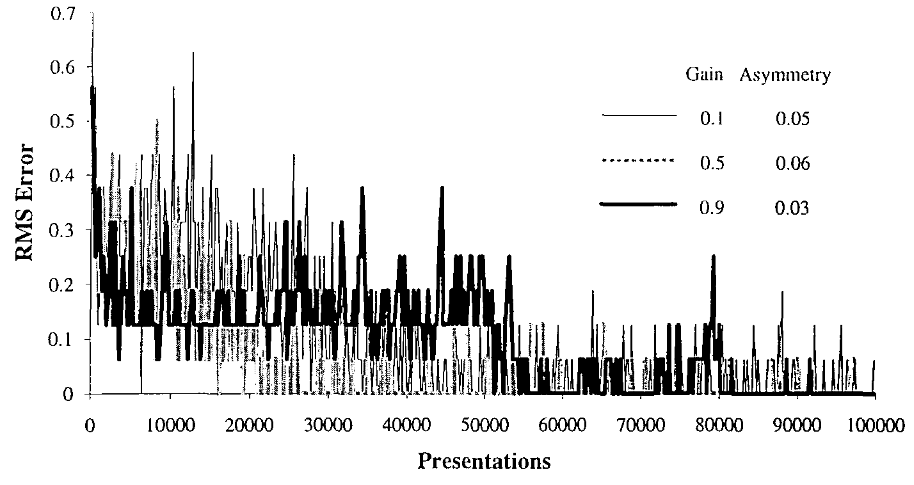


Figure 2.44: Q-model learning for the 4–2–4 encoder/decoder. *With a reduced problem size the Q-model is able to adapt to form the necessary weight values. The system can still get caught in an apparent local minima as exemplified by the plot for* $\rho = 0.9$ *and* $\lambda = 0.03$.

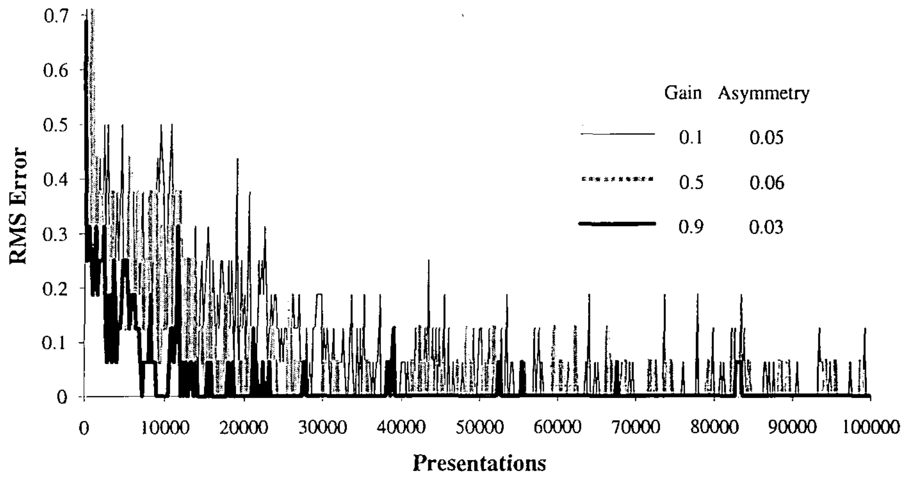


Figure 2.45: T-model learning for the 4–2–4 encoder/decoder. *As per the Q-model, Figure 2.44, with a reduced problem size the system is able to adapt to form the necessary weights to converge.*

Chapter 3

Hardware Implementation: A Critical Review

The previous chapter has discussed ANN architectures and the classes of learning algorithms which may be implemented. One of the problems which exists with many of these architectures and algorithms is that they exist only as mathematical models or are implemented as a software solution upon a standard von Neumann style architecture machine. The power of ANNs is derived from the high degree of parallelism that can be achieved. Despite the high speed of modern computer platforms for the simulation of ANNs, the platforms are often not fast enough for very large networks or real-time applications. The following difficulties, as highlighted by Atlas and Suziki [38] are to blame.

Massive interconnections can be required.

Most architectures involve tens, hundreds even thousands of neurons requiring interconnection. This is particularly acute in a fully connected NN. Each connection will require a multiplication and each neuron will therefore need many multiplications and summations of results.

Learning.

Many of the problems thought best suited to the solution by NNs have large data sets. Most algorithms are slow to converge to a solution due to adjusting the many weights that exist and this may necessitate many iterations.

Trial and error.

NNs do not always converge to a solution. When they do converge this may not be to a global minimum. Different training runs may be needed to be tried with various initial conditions to enable the best results to be selected.

Flexibility.

ANN algorithms and architectures are continuously evolving. A hardware solution must be as adaptable and adjustable as possible.

Therefore, it is worthwhile developing hardware realisations of ANN to increase the rate of processing and the size of problem which can be tackled in a rational timescale.

What possible systems are there for implementing an ANN in hardware? Analogue electronics, digital electronics, optical devices or any other system which may currently be in vogue. Points to be considered are the complexity of the resulting system (on top of the interconnectivity of the neurons), stability of the system, the ability of the system to learn on or off line.

3.1 Analogue Artificial Neural Networks

The basic operation of an ANN processing element as described in §2.2 can be summarised as

$$N_{OUT} = F\left[\sum(\Pi)\right]$$

therefore within analogue hardware it is necessary to perform the three operations of multiplication Π, summation $\sum$ and activation function F. Graf & Jackal [39] and Foo *et al* [40] provide a general introduction into analogue implementations, while Mead [41] provides a greater depth and more specialised viewpoint for using analogue circuits.

The basic instantiation of these three operations within an ANN is as follows

Multiplication. A single transistor could be used to perform multiplication, but a better approach would be to represent the strength of a connection by a resistor. In the latter case the output from a neuron i is input to a neuron j through a conductance representing the connection strength or weight T_{ij}. If the voltage at the input to neuron j is held at ground a current I_{ij} will flow through the conductance representing the weighted signal.

$$I_{ij} = V_{OUT_i}T_{ij}$$

The realisation of this weighting conductance can be achieved in several ways, including a CMOS switch operating in its active region, a switch-capacitor network, a switched-resistor network or a switched-ladder resistor network, all are illustrated in Figure 3.1.

Summation. The addition of input signals, currents, can be achieved by connecting the input wires together at a single node. An example would be the input of an operational amplifier (Op Amp) which is considered to be at virtual ground.

Pros	Cons
Speed of operation	Lack of thermal stability
Asynchronous behaviour	Low noise immunity
Easy implementations	Interconnection problems
Simple circuits	Limited accuracy
Small circuit elements	Hard to test
Direct interfacing	Basic components hard to fabricate
Basic storage of weights	Lack of design tools
Smooth neural activation function	Signed storage of weights
Massive parallelism	Non-uniform processing

Table 3.1: Implementation considerations for analogue neural networks

Activation Function. The format of activation realised will depend upon the configuration of the Op Amp at whose input the currents are summed. At the simplest level an Op Amp can be configured as an analogue comparator, a step function can thus be formed. A basic clipped linear activation function can be created using a non-inverting Op Amp configuration. Finally, a basic sigmoidal function may be achieved using two Op Amps in series. These three concepts are illustrated in Figure 3.2.

There are several advantages to following an analogue solution to hardware implementation, amongst them are the relatively simple circuits necessary, their small size and the ease with which they can be designed. This can lead to a high level of integration and a massively parallel design. As there does not need to be an overall clock to control the operation, this can be both fast and asynchronous. Finally, the connection strengths are represented by basic electronic components, eg. resistors and capacitors, no sophisticated circuit control mechanism is required.

However, analogue solutions are not without their problems. Analogue circuits lack thermal stability and have a low threshold to noise immunity. Despite being small and offering the possibility of a high level of parallelism how are the large number of connecting wires to be routed? The basic components which can be used for weight representation, resistors and capacitors, are hard to fabricate accurately and repeatedly. How are signed weights to be represented and stored? Analogue design tools for integrated circuits are not as well developed as their digital counterparts making the design of a circuit more difficult.

The pros and cons for the analogue implementation of ANN are summarised in Table 3.1

3.2 Digital Artificial Neural Networks

In a digital implementation of an ANN processing element it is obviously necessary to perform the same operations as with an analogue approach. A number of approaches can be taken to generating a network. One is to form all the components of a neuron separately using digital technology. A second is to generate digital architectures and processors tailored towards ANN implementation and application, ie. to design neurocomputer devices and accelerator boards. A third is to make use of existing high performance parallel computers and devices to construct purpose built machines, for example using transputers, or parallel DSP devices. Atlas & Suziki [38] provide a general introduction to digital NN systems.

Yet another approach using digital circuits is to use pulse coded computation as exemplified by Murray *et al* [27] with a deterministic approach and Tomlinson *et al* [42] and Leaver [43] with a stochastic approach. The pulse coded idea will be enlightened upon further in §3.4. .

Whichever of the above techniques is selected, digital technology has several consistent characteristics. The method of using binary data provides excellent noise immunity. The level of computation precision and accuracy does not depend upon the transistor size but on the number of bits used. The dynamic range of the system is influenced by the number of bits used. Digital circuits are relatively easy to design with many packages available for design and analysis before committing to silicon and the testing of the final fabricated product. Programmable components can be incorporated into a design to enable a system to be reconfigured by a software controller. Large matrices of synaptic weights can be stored in digital memory. Digital input/output can be multiplexed to reduce the number of physical connections both internally within a device and from device to device while maintaining a high level of connectivity for an overall network; this will of course be at the expense of an increase in complexity and a reduction in speed.

There are drawbacks to the use of digital hardware for the implementation of ANNs. Due to the switching action of transistors as devices operate and the constant charging/discharging of capacitors a higher power rating results. Digital circuits for addition, multiplication etc. are complex requiring many components and are expensive with respect to semiconductor usage. Despite the high level of integration that is possible and further advances in the reduction of device size the amount of semiconductor substrate required will be high. Digital processing at present is inherently a sequential operation leading to slower networks with respect to the number of interconnections per second which can be achieved. Finally, it must be remembered that the world is analogue in nature and an additional overhead of analogue to digital and digital to analogue conversion may need to be accounted for. It is likely that these conversions will only be upon the initial input and final output from the network and may not place too great an overhead upon performance.

The pros and cons for the digital implementation of ANNs are summarised in Table 3.2.

Pros	Cons
High noise immunity	Speed of operation
Precision	High component count
Existing design tools	High power dissipation
Programmable components are possible	A/D and D/A required
Store fixed and adaptive weights	Synchronous behaviour
High speed individual computations	Multiplexors occupy large area
Multiplex/Demultiplex	

Table 3.2: Implementation considerations for digital neural networks

3.3 Hybrid Artificial Neural Networks

A mixture of analogue and digital techniques for the hardware implementation of ANNs could be combined to provide a hybrid solution. This could lead to the best, or the worst features, of both disciplines being combined.

In a hybrid system weight storage and update can be performed digitally since this provides a more stable method than their analogue counterparts. Actual computation could be performed using analogue processing circuits as this often provides the smaller, faster circuits. Inter-element communication could be a mixture of digital and analogue. Analogue communication links could be used internally within an individual neural chip. Digital communication links could be used inter-chip or through a complete neural processing system.

Alternatively, pseudo analogue systems could be realised using digital signals by means of pulse encoding.

3.4 Pulse Coded Hardware Implementations

Digital encoding techniques for coding analogue information are highly developed especially for the field of communications. The aim in this section is to briefly describe methods and possible schemes for processing analogue signals as pulse sequences. It will be explained how the schemes offer several potential advantages over conventional analogue signal processing and numerical digital signal processing.

Pulse stream coded information has been implemented in several ways by various researchers into their application for neural networks. The neuron elements of these networks will be described.

Several pulse coding techniques exist for coding information into a pulse domain. These schemes can be divided into deterministic and stochastic methods which will be further elaborated on.

Pulse modulation techniques have been widely developed and include

- Pulse Width Modulation
- Pulse Position Modulation
- Pulse Amplitude Modulation
- Pulse Code Modulation
- Phase and Delay Modulation
- Pulse Frequency Modulation, or Deterministic Pulse Rate Encoding
- Stochastic Pulse Rate Encoding

With all these schemes the information is contained within the properties of the pulse or a specified group of pulses.

A complete description of most the above coding schemes can be found in Stremler, [44]. Three more pulse encoding schemes which are not described by Stremler are as follows:

Phase and Delay Modulation. Two output lines are required for this method. The signal is represented by the phase difference which occurs between the two lines. One line is a regular pulse stream while the delay of the pulses in the second line is relative to the first in proportion to the size of the signal.

Pulse Frequency Modulation, PFM, or Deterministic Pulse Rate Encoding. Pulses of constant amplitude and duration are generated but at a rate proportional to the signal. Within a given time period the signal can be deduced from the number of pulses received. For a specific signal level the pulses are produced in a regular deterministic manner.

Stochastic Pulse Rate Encoding. Pulses of constant width and amplitude are generated. The pulse sequence generated has the probability of a one appearing on the line proportional to the signal value to be encoded. Single line or dual line, unipolar and bipolar systems exist. These techniques are more fully discussed later in this thesis, §4.

The pulse encoding schemes described above have been developed for different environments. They are often most suited for the transmission of data and not necessarily the manipulation of data as required for numerical computation. This does not mean that calculations could not be achieved, rather that the schemes are not appropriate for these operations.

The basic desired numerical operations have already been outlined as addition and multiplication. Combining the pulse encoding schemes and numerical operations is not always satisfactorily achieved. PWM and PPM implementations of these operations are not known about although the design of suitable circuits is obviously feasible.

PAM signals may be used to perform these operations if the pulse sequences are synchronised. Analogue adders and multipliers operating upon the pulses may be used. Using the PA system would not offer any computational advantage when compared to complete analogue signal manipulation. Problems of stability and noise immunity for these operations exist. Improving these qualities increases the complexity of circuits. It would be necessary to maintain synchronism between the pulse streams.

PCM is suitable for numerical computation, particularly where a linear coding method is employed. Digital computers manipulating data encoded as binary information are all too common. Processing engines for addition, multiplication and other mathematical operations, eg. Fast Fourier Transforms, are highly developed. These implementations vary from the specific Digital Signal Processor, DSP, circuits, eg. Motorola DSP96002 or Texas Instruments TMS320 series, to the more general purpose implementations within microprocessors, eg. Intel 80x86 series or Motorola 68000 series. The basic building blocks for addition and multiplication are well known, the disadvantage is that the circuits are complex but their operation is consistent.

The use of stochastic pulse rate encoded sequences for numerical computation is surprisingly direct. Basic logic gates can be used to perform multiplication, addition and inversion. The accuracy of the result obtained depends upon the time taken to observe the output pulse stream since the information is represented as a probability or expected value.

3.4.1 Deterministic Pulse Coding Circuits

Much work has been conducted by Murray, at the University of Edinburgh, into the hardware implementation of NNs using deterministic encoding strategies.

The original system investigated was based upon asynchronous pulses, [45, 46]. The neuron could adopt one of the two states, on or off. When on and firing the output is a stream of pulses of fixed frequency and width. The pulses are generated by a ring oscillator. The parameters of the pulse stream are fixed by the time constants of the oscillator. As with many neuron circuits the condition as to whether or not to fire is based upon the weighted sum of inputs. Here the inputs are divided into excitatory and inhibitory pulse streams which both feed an integrator. If the excitatory pulses exceed the inhibitory ones the integrator charges up turning on the oscillator, else the integrator is discharged and the neuron does not fire.

The input pulse streams in a synapse are weighted deterministically using the contents of standard RAM. The MSB is the sign bit which determines if the pulses are to excite or inhibit the neuron. The remaining bits are used to gate the *Chopping Clock* signals which have Mark:Space ratios $1:1$, $1:3$, $1:7$, $\ldots$, $1:(2^{p-1}-1)$, where p is the number of bits in the weight. The pulses from the synapse are added to the overall pulse streams by using **OR** gates. It is not necessary for the pulse stream inputs to be synchronous for the neuron to operate, but the chopping clocks in the individual synapse circuits must be synchronous to obtain the correct weighting.

This topology does not provide for any learning in hardware. All training is performed off line and the weight RAM for the synapses loaded with the appropriate values.

The above idea proved unsatisfactory for a number of reasons. The digital weight storage required too large an area. The separate lines for the excitatory and the inhibitory pulse streams were considered clumsy and inefficient. The pseudo-clocks were not thought of as either aesthetically pleasing or smooth enough for dynamic behaviour.

A second system was designed in collaboration with the University of Oxford, [47, 48, 49, 50]. The level of neural activity is again represented by a regular pulse stream of fixed magnitude pulses. The rate of these pulses is dependent upon the level of neural activity as they are produced by a voltage controlled oscillator, VCO. The input to the VCO is from the sum of the synapse values.

The synapses are formed from MOST transconductance multipliers. These multipliers generate the product of two voltages as a current. One voltage input is the constant magnitude pulse stream from a previous level of varying frequency. The second voltage is the weight value to be applied to this pulse stream. This is an analogue voltage on a capacitor which is refreshed from a value stored externally on RAM. The resulting scaled pulses from each synapse will affect the charge accumulation on an integrator. The integrator voltage feeds the VCO of the neuron.

The basic neuron design is very simple and is able to produce an analogue output representation. Simulation and actual circuit fabrication have proved highly successful in the specific problem of position location for a robot.

With the signals being represented not only as the frequency of pulses but also as the amplitude of these pulses, how susceptible are they to analogue noise? How stably can the weight values on the capacitor be maintained? It must be admitted that these analogue values only exist locally within the neuron, the main signalling being a digital waveform.

A third mixed analogue digital pulse rate system has been presented by Murray *et al* recently, [51]. This system is specifically orientated towards a multi-layer perceptron configuration. The system varies from earlier ones in that the coding of information is in the pulse widths and that the system is synchronous. A constant pulse frequency is used which is controlled by a master clock. Computations occur during the first half of the cycle, the results are transmitted through a sigmoidal function during the second half of the cycle. The Mark:Space ratio of the pulses contains the neural state information. Fully on, 1.0, is represented by 1:1; fully off, 0.0, by no pulse at all; half on, 0.5, by a pulse of 1:3 Mark:Space ratio. Benefits of the system are the high throughput of calculations in conjunction with the parallel nature of the network. No learning has yet been incorporated into the network.

The above circuits and implementations can be found in Murray and Tarassenko's recent book, [27].

At the University of Kent, [52], a neural circuit has been designed which uses an analogue voltage input and produces an analogue output voltage. The neuron conducts internal processing using pulse streams. The pulse streams for each signal are asynchronous. Analogue inputs to the neuron are converted to pulse streams of fixed width but variable frequency by a VCO. Weighting of these pulse streams is achieved by PWM. The resulting weighted pulses are summed using an **OR** gate before integrating the total, so forming an analogue output voltage. The neuron is designed so that the maximum Mark:Space ratio of the input pulse stream is 1:10. After weighting the Mark:Space ratio value will be reduced. The incidence of coincident pulses at the summing **OR** gate will be low. An inhibition signal is applied to the resultant pulse stream before integration, again this is carried out by PWM. The problem of weight storage was not resolved, the possibility of external RAM refreshing an analogue voltage on the gate of a transistor was stated. No on-line learning was presented. Frequency of operation of the circuits was high to reduce the RC component values in the timing sections of the neuron. This had the bonus of keeping a high throughput of data. Maintaining consistent and stable timing using the RC time constants was a problem with the idea.

3.4.2 Stochastic Pulse Coding Circuits

The previous section concentrated on work which used regular pulse streams to perform computation. In this section an overview of some neural circuit implementations based upon stochastic pulse encoding techniques is presented. The mechanics of this style of encoding, computation and decoding are fully discussed in the following chapter, §4. The possibility of using stochastic pulse systems for NNs was highlighted by Gaines [53].

An associative memory neural network simulation was reported by Nguyen and Holt, [54], in which stochastic processing elements were used. Encoding of signals used a pseudo-noise source formed from a Pseudo Random Binary Sequence, PRBS, shift register configuration. They highlighted the advantage of a stochastic implementation in terms of a low gate count, easier routeing of signals in parallel and improved noise immunity, the penalty being an increase in processing time compared to the direct DSP implementations. One reason is that the results are gained by time averaging the output pulses. As a network grows the multiplier of a DSP chip would become an increasing bottleneck reducing the speed differential. The accuracy of Nguyen and Holt's system was comparable with a 10-bit digital parallel multiplier.

A stochastic implementation of a Hopfield net has been achieved by Van Den Bout and Miller, [55, 56]. This design made extensive use of shift registers and counters which occupied a significant amount of silicon. The design was expandable to allow the Hopfield net to grow to larger sizes. Two interesting points were raised by this work. First, the dynamic range of the weights could be increased by use of an exponential distribution of the random numbers used to encode them. This will lead to a logarithmic distribution of weight values. Computational circuits are unaffected since it is the interpretation placed upon the resulting pulse streams which is important. Second, by adjustment of the Probability Density Function, PDF, for the random number generator controlling the output of the neuron circuit, the output function can be varied. A uniform PDF will produce a linear transfer function with hard limits, a sigmoidal transfer function can be achieved by using a Gaussian distribution.

Investigation of a stochastic neural circuit has been conducted by Banzhaf, [57], Figure 3.3. The neurons made use of **AND** and **OR** gates for computation. The aim was to realise primitive neuron-type functions, not to perform accurate algebraic manipulation. This was evident mainly in the performance of addition by use of a single **OR** gate, as pulses became more dense and the result less accurate, the output begins to saturate at unity. By implementing a gate structure which allowed excitatory and inhibitory signals, a sigmoid style non-linearity could be formed. The effect of representation of weight pulses was assessed. The weighting pulses were produced on different time scales and with different quantities of dead-time. The latter point could cause synaptic gates to operate near to their points of instability.

Tomlinson *et al* [42] discuss a stochastic pulse rate NN implementation system which was subsequently fabricated into a chip set, the Neural Semiconductor SU3232 and NU32. Similar to Banzhaf above inexact summation of the excitatory and inhibitory net input is performed but this time a **WIRED-OR** is utilised. The **WIRED-OR** conserves on chip substrate area and allows scalable summation of many inputs to be performed. Eguchi *et al* [58, 59] also use the ideas of Tomlinson *et al* to produce their experimental NN chip.

Kondo *et al* [60] utilised stochastically encoded data in their two proposed architectures of Figure 3.4 and Figure 3.5. Their first proposal, Figure 3.4, iteratively cycles through each input and associated weight before generating an output pulse. The weighted input value pulses are summed in an up/down counter before passing through a sigmoid transform. Their second proposal, Figure 3.5, weights each input in parallel before performing an analogue summation of the resultant values. The result of the summation is then passed through a sigmoid transform. In both designs it is interesting to note that the sigmoid transform is performed by comparison of the weighted sum of inputs with a Gaussian random number. This technique will be returned to and developed in §4.7 using an entirely digital circuit.

A thesis by Hyland, [61], investigated the use of stochastic pulse encoding and computation to a particular type of model for neural networks, the Boltzmann Machine. Several encoding systems were discussed and simulated. Hyland's tests mirrored Ackley's, Hinton's and Sejnowski's, [31], original experiments. Learning of the 4–2–4, 4–3–4 and 8–3–8 encoder mappings was achieved with varying degrees of success. Due to the simulation being conducted on a serial computer rather than a parallel network or a dedicated hardware configuration, Hyland found the processing to be exceedingly labourious. The requirement to use specific hardware for improved performance was evident.

3.5 Commercial Hardware Realisations

Few commercial hardware realisations of dedicated neurons or network devices have been produced and marketed. Devices which have been include the ETANN and Ni1000 by Intel, SU3232 and NU32 chip set by Neural Semiconductors, the NiSP by MCE and finally the NEURO4 by Mitsubishi. There are many forms of accelerator boards which incorporate DSP chips eg. TMS320C40 or fast co-processors eg. i860, which have been produced together with supporting software libraries for driving these systems. These boards are of a more general purpose nature and not necessarily to be used for NN applications.

ETANN

The ETANN, Electronically Trainable Analog Neural Network, [62, 63], produced by Intel is an analogue device consisting of 64 neurons. No on-chip learning is provided for the device, instead all learning and training is conducted off-line using third party development systems hosted on a PC, eg. iDynaMind by NeuroDynamX or iBrainMaker by California Scientific Software. Neuron weights are downloaded to program the device once adaption has taken place.

Ni1000

The Ni1000 is another device NN device developed by Intel, [64]. Unlike the previously developed ETANN this device is digital with a resolution of 5-bits. The Ni1000 has a maximum 256 input vectors which it is able to classify into 64 groups by means of a Radial Basis Function style algorithms. Operating several of these devices together will allow the number of degrees of classification to be increased. The Ni1000 has been integrated into an accelerator board by Nestor Inc., which together with their emulation software allows the development of NN based systems,

NU32/SU3232 Chip Set

Rather than produce a unified device Neural Semiconductors produced a set of devices, NU32 and SU3232. The SU3232 is a matrix multiplier with 32 inputs. There are 1024 weights in the device organised as a 32x32 weight matrix. The output function for a neuron is incorporated in the NU32 device member of the set. The format of computation used by Neural Semiconductor is a stochastic pulse rate method as described previously and which they refer to as Digital Neural Network Architecture, DNNA.[1]

[1]DNNA is a trademark of Neural Semiconductors, Inc.

NiSP

The NiSP (Neural Instruction Set Processor) is a RISC based processor designed specifically for NN operation, [65]. The device has an overall 12-bit data resolution and can have any desired activation function loaded into it. The device is optimised for feedforward network operation with only seven instructions in its entire instruction set. The size of feedforward network both in terms of the number of neurons and layers is limited by the amount of RAM connected to the processor which is 32k. The device is aimed at the embedded control system market, but as with all the above mentioned devices, a development board and emulation software is available.

NEURO4

Limited information is available on this device from Mitsubishi, but the device is digital containing 12 processors. The NEURO4 processor operates using 24 bit floating point representation. Currently the device is available in sets of four chips configured upon an accelerator board suitable for driving from a workstation. In addition the device can be used as an external set of processors for general purpose parallel processing.

3.6 Conclusions

In this chapter a review of the requirements for a hardware implementation of an artificial neuron or an ANN have been specified which include a high level of interconnectivity, small neuron size, ability for the neuron weights to be adapted on-line ie. the neuron to be trainable in a hardware implementation. It has been shown that the two principal approaches of analogue or digital circuitry may be used to formulate a neuron with sample circuits shown where relevant. The benefits and drawbacks of these two methods have been tabulated. A possible compromise may be a hybrid of the two approaches.

The techniques of pulse processing have been highlighted. Pulse processing is essentially a digital process but may be used to represent analogue values by varying pulse width, amplitude or frequency. The many and varied deterministic approaches adopted by Murray *et al* have been reviewed. Additionally, stochastic pulse rate encoding implementations by many researchers have been reviewed. These stochastic approaches have often been found to be deficient in a particular area eg. they perform inexact computation or move out of the digital domain for certain sections of their circuitry.

A hardware stochastic pulse rate computation approach would seem beneficial due to the ease of connectivity of the neuron, the potential simplicity of the circuitry and their improved immunity to noise compared to alternative systems. In the following chapter, §4, a thorough review of stochastic pulse rate encoding and processing techniques is conducted. New novel circuits are presented to maintain the accuracy of computation and to ensure that all the processing for an artificial neuron is kept within the digital stochastic pulse rate encoded domain. These circuits will then enable a hardware neuron to be designed and fabricated as described in Chapter 6. This neuron should also have the ability to have its weights, and therefore its performance, adjusted as a network is running. Demonstartion of the processing capability of the new hardware neuron will be provided by implementing a basic network for a simple test problem, the 4–2–4 encoder/decoder.

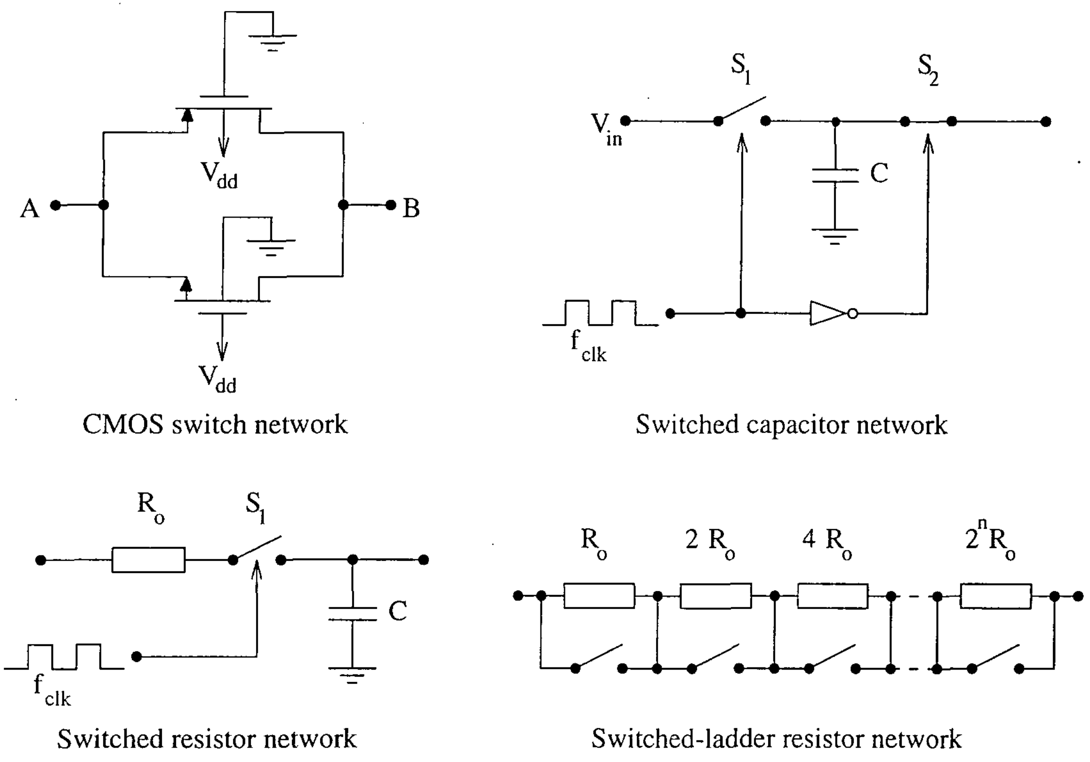


Figure 3.1: Example weighting conductance circuit configurations. *Note the simplicity of the circuits and the small number of components required.*

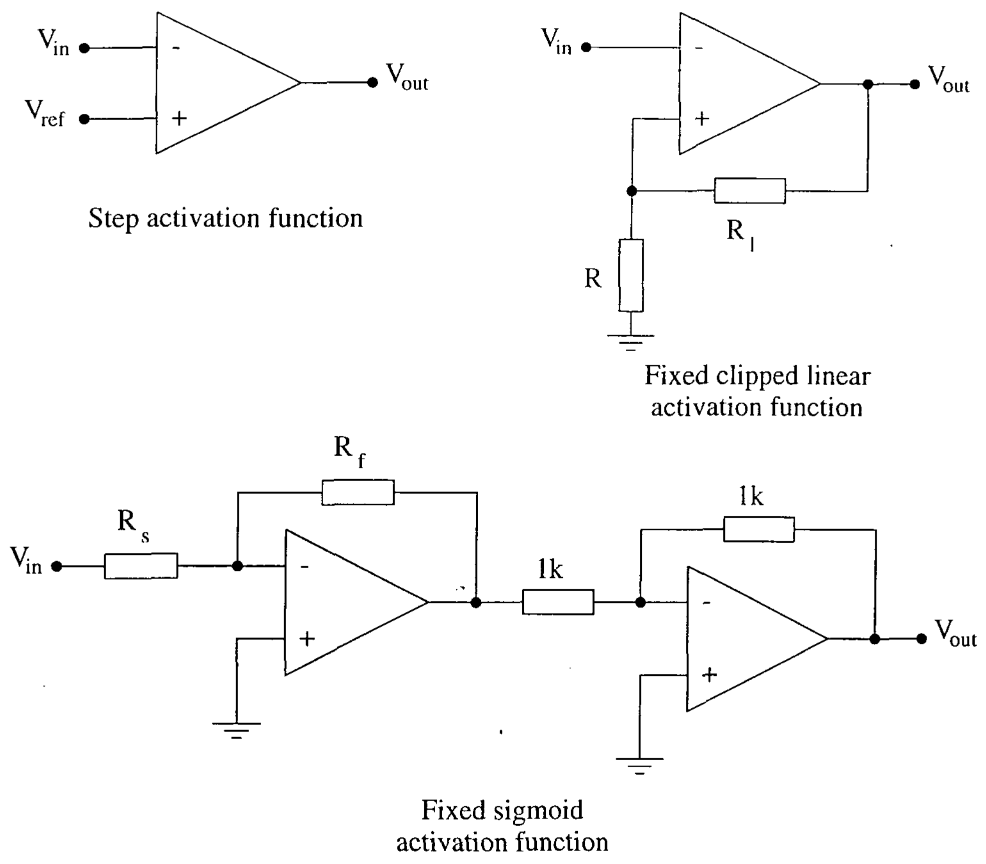


Figure 3.2: Example activation function circuit configurations. *As per Figure 3.1 note the simplicity of the circuits and the low component count.*

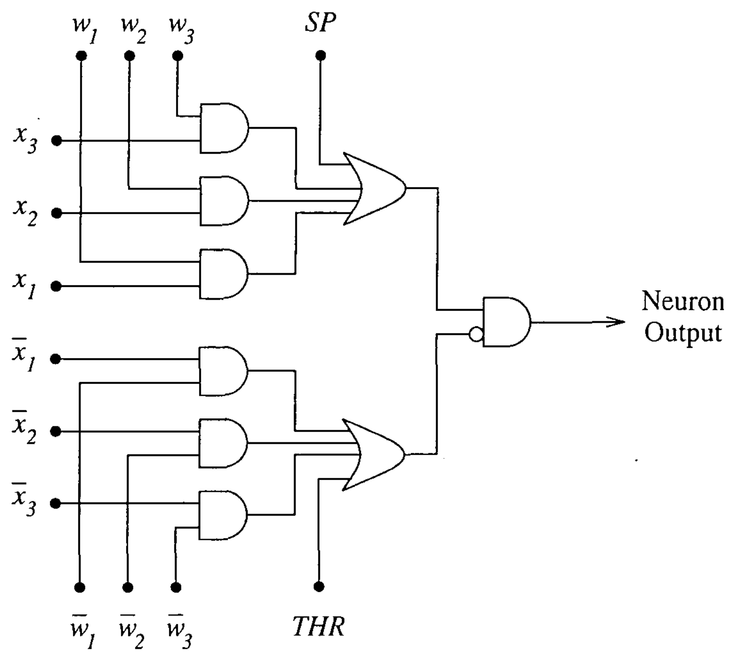


Figure 3.3: Banzhaf's stochastic neuron layout with excitatory and inhibitory inputs.

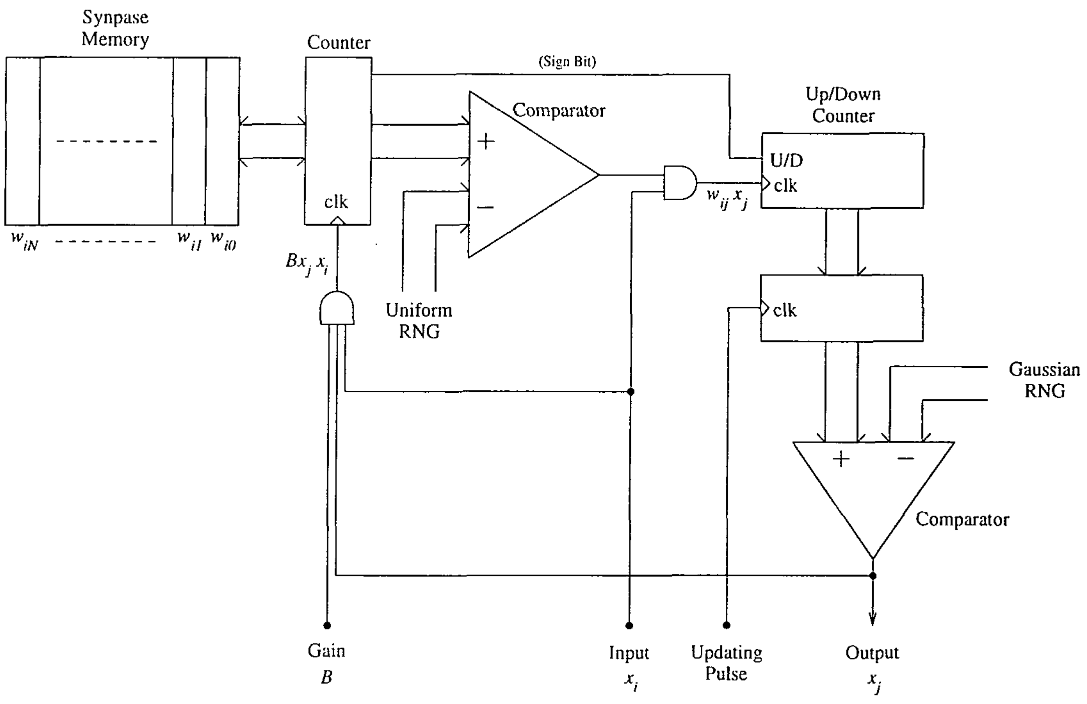


Figure 3.4: Kondo's first proposal. *Serial weighting of the inputs is performed with the result accumulated in an up/down counter. The more inputs there are to the neuron the longer it will take to realise an output pulse. Note how the sigmoid transform is performed by comparison with a Gaussian random number.*

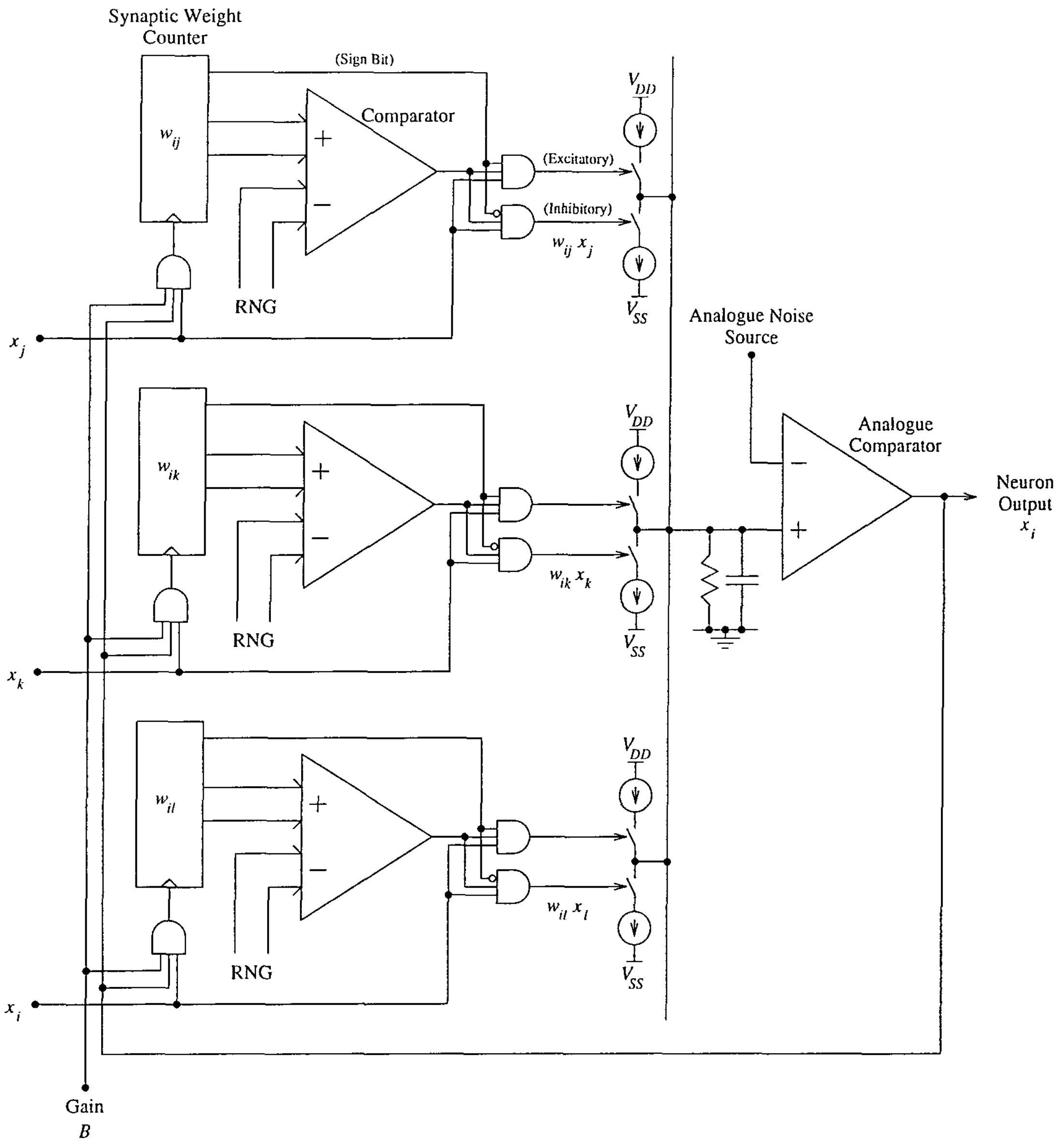


Figure 3.5: Kondo's second proposal. *Parallel weighting of the inputs occurs in this design, but the operation moves out of the digital into the analogue domain for summing these values. Again the sigmoid transform is performed by comparison with a Gaussian random number.*

Chapter 4

Stochastic Pulse Rate Computation

In earlier chapters of this thesis the broad concepts of ANNs have been introduced. In particular Chapter 2 made reference to several architectures and algorithms namely, the MLP, the Kohonen self-organising feature map, the Hopfield network and the Boltzmann machine. Besides software models and simulations hardware concepts for the implementation of ANNs have been reviewed in Chapter 3. From the review of hardware it can be seen that a need exists for a hardware implementation system that is cheap to construct ie. requires few components and uses non-complex fabrication techniques, is stable and accurate with respect to the storage of interconnection weight values, may be easily reprogrammed to perform a new task and finally the interconnection weight values may be easily adjusted by a learning scheme which is operating on-line. So far most of the hardware approaches offered are deficient in one or several of these areas.

Pulse rate computation has been proposed for hardware implementation to gain the benefit of both the analogue and digital worlds. Murray *et al* [45, 46, 66, 47, 48, 67, 50, 51, 27], Meador *et al* [68], Cotter *et al* [69], Tomberg *et al* [70] and Daniell *et al* [52] adopt a deterministic approach whereby communication and processing can be effected by using deterministic pulse sequences. Nguyen *et al* [54], Eguchi *et al* [58, 59], Tomlinson *et al* [42], Banzhaf [57] and Kondo *et al* [60] have followed a stochastic pulse rate encoded sequence policy. These proposals have involved analogue circuit forms or have performed inexact computations. The pulse rate method, in particular the stochastic pulse rate methods, are attractive since there is biological evidence that neurons signal via stochastic pulse streams, for example see Churchland *et al* [71].

If use is to be made of stochastic pulse rate encoding and computation techniques, it is first necessary to understand the operation of the basic component parts and why they will be of benefit. A critical review follows of stochastic encoding techniques, transferring information from a deterministic value into a stochastic pulse stream representation. Circuits are presented to perform multiplication, addition, subtraction and function approximation. New circuits are proposed for single line unipolar subtraction but more importantly the addition of N bipolar signals with an exact result. With the aim of designing an artificial neuron operating by use of these techniques it is necessary to derive an appropriate circuit for performing a non-linear transformation. The non-linearity circuit developed performs a sigmoidal transformation in the stochastic pulse rate encoded domain.

The techniques of stochastic pulse rate encoding and computation were first committed to paper in 1965 both by researchers at the Standard Telecommunications Laboratories [72, 73] and at the University of Illinois [74]. The technique relies upon the principle that the probability of a binary variable being a one is a representation of the required analogue information. In general, observing a signal at an instant will only produce an expected value result. To gain an increasingly accurate value it is necessary to average the number of pulses received over a given number of time slots. Several problems arise immediately, firstly, how is information translated into this domain? Secondly, how can negative numbers be accounted for? Finally, how can pulse streams be manipulated to perform mathematical computation. The input encoding strategies will be demonstrated first before considering the mathematics which may be performed.

4.1 Encoding or Input Mapping into the Stochastic Pulse Rate Domain

Several encoding strategies are put forward by Gaines [53] and Mars & Poppelbaum [75] including linear or non-linear mappings, unipolar or bipolar signals and whether one or two lines are to be used to transmit information between computation elements. The basic principles of input mapping can be understood by reference to three linear schemes, the simple Single Line Unipolar (SLU) strategy which will be developed into the Dual Line Bipolar (DLB) and finally Single Line Bipolar (SLB) strategies. Non-linear schemes for encoding with an infinite range in at least one direction will be briefly presented.

4.1.1 SLU Input Mapping

Given an input value x within the range $0 \leq x \leq X$ which it is desired to represent upon a single line as the probability of observing a pulse, a binary variable x_b may be defined with a generating probability p by the following transform.

$$p = p(x_b = 1) = \frac{x}{X}$$

Thus X, the upper boundary limit, will be represented by a signal which is always ON, and zero the lower boundary limit, will be represented by a signal which is always OFF.

To actually generate a binary pulse train of x_b's to represent x, x would be normalised by dividing by X and the resultant compared with a uniform normalised noise source n, $0 \leq n \leq 1$. If $x > n$ a one is produced as an output else a zero is produced. The comparison is undertaken at regular clock intervals so producing a stochastic pulse train. By this formation x_b is seen to be a Bernoulli random variable [76]. Figure 4.1 demonstrates an example of two values of x encoded as stochastic pulse streams.

Analysing the characteristics of the Bernoulli sequence, the value of x_b may be noted at each of the N clock intervals. Denoting the sample as x_{b_i} for that at the i'th clock pulse, an estimate of the generating probability $\hat{p}$ is

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} x_{b_i}$$

The expected value of this estimate is

$$\mathrm{Exp}\,[\hat{p}] = p$$

as would be expected for a Bernoulli sequence ie. the expected value is the original generating probability and is independent of the number of samples N taken. A Bernoulli sequence is a zero-order Markov chain. The accuracy of this estimate is a function of the number of samples taken and is determined by the variance of the expected value $\mathrm{Var}(\hat{p})$.[1]

$$\begin{aligned}
\mathrm{Var}\,(\hat{p}) &= \mathrm{Exp}\,[(\hat{p}-p)^2] \\
&= \mathrm{Exp}\left[\left(\frac{1}{N}\sum_{i=1}^{N} x_{b_i} - p\right)^2\right] \\
&= \mathrm{Exp}\left[\frac{1}{N^2}\left(\sum_{i=1}^{N} x_{b_i}\right)^2 - \frac{2}{N}\sum_{i=1}^{N} x_{b_i} p + p^2\right] \\
&= \frac{1}{N^2}\mathrm{Exp}\left[\left(\sum_{i=1}^{N} x_{b_i}\right)^2\right] - 2p^2 + p^2
\end{aligned} \tag{4.1}$$

Now,

$$\frac{1}{N^2}\left(\sum_{i=1}^{N} x_{b_i}\right)^2 = \hat{p}^2 = \frac{1}{N^2}\sum_{\forall i,j}^{N} x_{b_i} x_{b_j}$$

[1]The variance of a value A measures the expected square of the deviation of A from its expected value.

$$N^2\hat{p}^2 = \sum_{i}^{N} {x_{b_i}}^2 + \sum_{i\neq j}^{N} x_{b_i}x_{b_j}$$
$$= \sum_{i}^{N} {x_{b_i}}^2 + 2\sum_{i<j}^{N} x_{b_i}x_{b_j}$$

$$\text{Exp}\,[N^2\hat{p}^2] = N^2\text{Exp}\,[\hat{p}^2]$$

$$N^2\text{Exp}\,[\hat{p}^2] = \text{Exp}\,[\sum_{i}^{N} {x_{b_i}}^2 + 2\sum_{i<j}^{N} x_{b_i}x_{b_j}]$$
$$= \sum_{i}^{N} \text{Exp}\,[{x_{b_i}}^2] + 2\sum_{i<j}^{N} \text{Exp}\,[x_{b_i}]\text{Exp}\,[x_{b_j}]$$

$$x_{b_i} = \begin{cases} 0 \\ 1 \end{cases} \Rightarrow {x_{b_i}}^2 = x_{b_i}$$

therefore

$$N^2\text{Exp}\,[\hat{p}^2] = Np + {}_nC_2\text{Exp}\,[x_{b_i}]\text{Exp}\,[x_{b_j}]$$
$$= Np + N(N-1)p^2 \tag{4.2}$$

$$\text{Exp}\,[\hat{p}^2] = \frac{p + (N-1)p^2}{N}$$

Replacing eq.(4.2) in eq.(4.1)

$$\text{Exp}\,[\hat{p} - p]^2 = \frac{p + (N-1)p^2 - np^2}{N}$$

$$\text{Var}\,(\hat{p}) = \frac{p(1-p)}{N} \tag{4.3}$$

This leads to a standard deviation for $\hat{p}$ of

$$\sigma(\hat{p}) = \left(\frac{p(1-p)}{N}\right)^{\frac{1}{2}}$$

The expected error is zero for $p = 0$ or $p = 1$, and reaches its maximum value at $p = 0.5$ as illustrated on Figure 4.2. This diagram also illustrates the balance between accuracy and speed of determining the value of p. The more accurate a result required the more samples need to be averaged and therefore the longer it will take. Further effects of the time averaging period for converting from a stochastic pulse sequence to a deterministic signal are discussed in §4.8

4.1.2 DLB Input Encoding

Both positive and negative values of x, $-X \leq x \leq X$, can be represented by extending the SLU case to that using two lines, one line upon which positive values are encoded, the UP line (U) and the other line upon which negative values are encoded, the DOWN line (D). This can be accomplished by defining

$$p(U=1) - p(D=1) = \frac{x}{X} \tag{4.4}$$

No unique association exists between the probabilities represented by each line and the overall value represented . This is because there are two signal lines with a possibility of 4 signal conditions being used to represent a single value and for example an overall value of 0.6 can be represented by an UP line value of 0.6 and a DOWN line value of 0.0 or an UP line value of 0.8 and a DOWN line value of 0.2. The former case is known as the minimum variance form. Very distinct polarised starting pulse sequences can be defined with positive values only on the UP line and negative values only on the DOWN line.

Each pulse sequence in this dual line case is defined independently as follows for the minimum variance form of the value.

$$\begin{array}{l} x > 0 \Rightarrow \begin{cases} p(U=1) = \left|\dfrac{x}{X}\right| \\ p(D=1) = 0 \end{cases} \\ x < 0 \Rightarrow \begin{cases} p(U=1) = 0 \\ p(D=1) = \left|\dfrac{x}{X}\right| \end{cases} \\ x = 0 \Rightarrow \begin{cases} p(U=1) = 0 \\ p(D=1) = 0 \end{cases} \end{array}$$

For the purposes of analysis the following four values are defined

$$p(U=0, D=0) = v$$

$$p(U=1, D=0) = u$$

$$p(U=0, D=1) = d$$

$$p(U=1, D=1) = c$$

which obviously leads to

$$c + d + u + v = 1$$

We have

$$p(U=1) = u + c$$

$$p(D=1) = d + c$$

by eq.(4.4)

$$\Rightarrow u - d = \frac{x}{X} \tag{4.5}$$

If both the UP line and DOWN line are in the same condition this will correspond to zero and will not contribute towards the resultant.

For the DLB system the mean and variance may be obtained using a three-level random value B_i at the i'th clock pulse.

$$B_i = \begin{cases} 1 \\ 0 \\ -1 \end{cases}$$

$$B_i = U_i - D_i$$

where $U_i = 1$ for the UP line on and $D_i = 1$ for the DOWN line on. After N clock pulses the mean value of B_i is

$$\hat{B} = v.0 + u.1 + d. - 1 + c.0$$

$$\hat{B} = u - d = \frac{x}{X}$$

The variance of $\hat{B}$ is determined by

$$\mathrm{Var}\,(\hat{B}) = \frac{\mathrm{Exp}\,[{B_i}^2] - \hat{B}^2}{N}$$

$$\begin{aligned} \mathrm{Var}\,(\hat{B}) &= \frac{v.0 + u.1 + d.1 + c.0 - (u-d)^2}{N} \\ &= \frac{u + d - (u-d)^2}{N} \\ &= \frac{u(1-u) + d(1-d) + 2ud}{N} \end{aligned} \tag{4.6}$$

It can be seen that the variance is minimised if either $d = 0$, $(u > d)$ or $u = 0$, $(u < d)$ and leads to the minimum variance mapping

$$p(U=1) = \begin{cases} \frac{x}{X} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$p(D=1) = \begin{cases} 0 & x > 0 \\ -\frac{x}{X} & x \leq 0 \end{cases}$$

A unique probability for c, d, u and v does not exist due to the equivalence of $(U = 1, D =$

1) both on and ($U = 0, D = 0$) both off. If it is assumed both lines are never on together (simple gating can ensure this in practice) then $c = 0$.

4.1.3 SLB Input Encoding

The final linear transformation scheme to be considered is that of representing bipolar quantities on a single line. For an input value x, $-X \leq x \leq X$, the binary variable x_b with a generating probability p, the following transform is used,

$$p = p(x_b = 1) = \frac{x}{2X} + \frac{1}{2} \tag{4.7}$$

Maximum positive value, X, is given by a logic level of always on, maximum negative value, $-X$, by a logic level of always off and zero by a random fluctuating logic level with an equal probability of being either on or off.

If $\hat{p}$ is an estimate of p as for the SLU case then

$$\hat{p} = \frac{\hat{x}}{2X} + \frac{1}{2} \tag{4.8}$$

$$\frac{\hat{x}}{X} = 2\hat{p} - 1 \tag{4.9}$$

The variance of this estimate may be gained in the following manner.

$$\mathrm{Var}\left(\frac{\hat{x}}{X}\right) = \mathrm{Var}\,(2\hat{p} - 1)$$

For two independent random variables R and S

$$\mathrm{Var}\,(R + S) = \mathrm{Var}\,(R) + \mathrm{Var}\,(S)$$

therefore

$$\begin{aligned}\mathrm{Var}\left(\frac{\hat{x}}{X}\right) &= \mathrm{Var}\,(2\hat{p}) - \mathrm{Var}\,(1) \\ &= \mathrm{Exp}\,[(2\hat{p} - 2p)^2] \\ &= 4\mathrm{Exp}\,[(\hat{p} - p)^2]\end{aligned}$$

which by use of eq.(4.3)

$$\mathrm{Var}\left(\frac{\hat{x}}{X}\right) = \frac{4p(1-p)}{N}$$

which by use of eq.(4.7) is

$$\mathrm{Var}\left(\frac{\hat{x}}{X}\right) = \frac{1 - (\frac{x}{X})^2}{N} \tag{4.10}$$

The variance of the estimate of x is zero for maximum positive and negative values but a maximum for $x = 0$.

4.1.4 Non-linear Input Encoding

The transforms listed in the above three sections have been linear transforms with a finite range of values which may be encoded. For completeness there now follows some examples of non-linear transforms which have an infinite range in at least one direction. No analysis of variance is presented as the schemes are shown for information only.

Using a single line an input range $0 \leq x \leq +\infty$ can be encoded as a probability p of observing a one on the line as

$$p = \frac{x}{e + x}$$

e is defined as the centre value for encoding, it is the point at which $p = 0.5$.

$$x \to 0 \quad \Rightarrow p \to 0$$

$$x = e \quad \Rightarrow p = 0.5$$

$$x \to +\infty \Rightarrow p \to 1$$

For $x < e$ the value of p will vary rapidly, but for $x > e$ the probability varies more slowly. Figure 4.3 shows a sample transformation for $e = 5$. The effect of varying e is to alter the position of the 'knee' of the transformation curve. To retrieve values from the stochastic domain

$$x = \frac{ep}{1 - p}$$

Bipolar values of x in the input range $-\infty \leq x \leq \infty$ can be encoded onto a single line by

$$p = \frac{x - e + \sqrt{(x^2 + e^2)}}{2x}$$

not a simple transform. Decoding is achieved by

$$x = \frac{e(1 - 2p)}{2p(p - 1)}$$

This scheme allows completely arbitrary values to be encoded into the stochastic pulse domain and is also illustrated in Figure 4.3 for a value $e = 5$. The effect of varying e is to alter the gradient of the transformation curve.

Having reviewed the main forms and principles of stochastic pulse rate encoding the basic mathematical operations of inversion (negation), multiplication and addition will now be presented together with Boolean logic circuits to perform the required tasks in hardware. Only the linear encoding schemes will be considered. Due to the complexity of input encoding and decoding for the non-linear strategies they will not be considered.

4.2 Inversion

Inversion, negation or complementation can be achieved by using at most a single logical inverter for the three linear encoding schemes. For the SLU and SLB a single logical inverter in the line will suffice, while for the DLB case merely exchanging the two signal lines performs the necessary action, ie. UP $\rightarrow$ DOWN and DOWN $\rightarrow$ UP, Figure 4.4

In the SLU case the inverter complements the input sequence x_i so that the output x_o is

$$x_o = 1 - x_1$$

$$\text{Exp}\,[x_o] = \text{Exp}\,[1 - x_1] = 1 - \text{Exp}\,[x_1]$$

$$p_o = 1 - p_1$$

a trivial result.

In the DLB case where the two lines are exchanged

$$\begin{aligned}\text{Exp}\,[x_1] &= \text{Exp}\,[x_1^U] - \text{Exp}\,[x_1^D] \\ &= p(U_1) - p(D_1) \\ \text{Exp}\,[x_o] &= \text{Exp}\,[x_o^U] - \text{Exp}\,[x_o^D] \\ &= p(D_1) - p(U_1) \\ \Rightarrow\ x_o &= -x_1\end{aligned}$$

The output inverted signal is equivalent to the negative of the input signal.

In the SLB case

$$x_o = 1 - x_1$$

$$\text{Exp}\,[x_o] = 1 - \text{Exp}\,[x_1]$$

$$p_o = 1 - p_1$$

as for the SLU above but,

$$p_i = \frac{1}{2} + \frac{x_i}{2X}$$

$$\frac{1}{2} + \frac{x_o}{2X} = 1 - \left(\frac{1}{2} + \frac{x_1}{2X}\right)$$

$$x_o = -x_1$$

and the output signal is the negative equivalent of the input signal.

4.3 Multiplication

Taking each of the three linear encoding schemes in turn it will be demonstrated how Boolean logic gates may be used to achieve the multiplication of two stochastic pulse streams.

For the SLU case with two input streams p_1 and p_2 an **AND** will perform multiplication to generate the output p_o.

$$p_o = p_1 p_2$$

when

$$p_i = \frac{x_i}{X}$$

therefore

$$\frac{x_o}{X} = \frac{x_1 x_2}{X.X}$$

$$x_o = \frac{x_1 x_2}{X}$$

The normalised product of inputs x_1 and x_2 with respect to the range of X is found. This is always representable.

The variance of this product $\mathrm{Var}\,(p_o)$ is obtained by using eq.(4.3),

$$\mathrm{Var}\,(\hat{p}_i) = \frac{p_i(1-p_i)}{N}$$

thus

$$\mathrm{Var}\,(\hat{p}_o) = \frac{p_1 p_2(1-p_1 p_2)}{N} \tag{4.11}$$

this can be verified to be

$$\mathrm{Var}\,(\hat{p}_o) = p_1 \mathrm{Var}\,(p_2) + p_2 \mathrm{Var}\,(p_1) - N\mathrm{Var}\,(p_1)\mathrm{Var}\,(p_2) \tag{4.12}$$

The equivalence of eq.(4.11) and eq.(4.12) can be demonstrated by expansion of eq.(4.12).

For the DLB representation it is necessary that two positive or two negative quantities produce a positive result which implies that when both the UP inputs are on or both the DOWN inputs are on the output UP should be on. However, if an UP and a DOWN are on together the the output DOWN must be on. Figure 4.5 demonstrates the required gating arrangement. Using the previously defined probabilities for a dual line system (v,

u, d and c) the output probabilities of the multiplier are given by

$$\begin{aligned} v_o &= v_1 + v_2 - v_1 v_2 \\ u_o &= u_1 u_2 + d_1 d_2 \\ d_o &= u_1 d_2 - d_1 u_2 \\ c_o &= c_1(1 - v_2) + c_2(1 - v_1) - c_1 c_2 \end{aligned} \tag{4.13}$$

therefore

$$\begin{aligned} u_o - d_o &= u_1 u_2 + d_1 d_2 - (u_1 d_2 - d_1 u_2) \\ &= (u_1 - d_1)(u_2 - d_2) \end{aligned}$$

By using eq.(4.5)

$$\frac{x_i}{X} = u_i - d_i$$

we obtain

$$x_o = \frac{x_1 x_2}{X}$$

Given that both the input values to the multiplier are in the minimum-variance format it is possible for only one at most of the following terms to be non-zero, u_1u_2, d_1d_2, u_1d_2 or d_1u_2. By inspection of eq.(4.13) it can be seen that only one of u_o or d_o may be non-zero and thus the resultant of the multiplier will be in minimum-variance format. From eq.(4.6)

$$\begin{aligned} \mathrm{Var}\left(\frac{\hat{x}_i}{X}\right) &= \frac{u_i + d_i - (u_i - d_i)^2}{N} \\ \Rightarrow \mathrm{Var}\left(\frac{\hat{x}_o}{X}\right) &= \frac{u_o + d_o - (u_o - d_o)^2}{N} \end{aligned} \tag{4.14}$$

this can be shown to be

$$\mathrm{Var}\left(\frac{\hat{x}_o}{X}\right) = (u_1 + d_1)\mathrm{Var}\left(\frac{\hat{x}_2}{X}\right) + (u_2 + d_2)\mathrm{Var}\left(\frac{\hat{x}_1}{X}\right) - N\mathrm{Var}\left(\frac{\hat{x}_1}{X}\right)\mathrm{Var}\left(\frac{\hat{x}_2}{X}\right) \tag{4.15}$$

by expanding fully both eq.(4.14) and eq.(4.15).

For the SLB representation the gating is required to produce an output pulse when both signal lines are in the same state, both on or both off and no signal when the two input lines are different states. An appropriate circuit is shown in Figure 4.6. The circuit can be recognised as an **XNOR** gate.

The output generating probability p_o can be expressed in terms of the two input generating probabilities.

$$p_o = p_1 p_2 + (1 - p_1)(1 - p_2)$$

since

$$\text{Exp}[x_o] = \text{Exp}[x_1 x_2 + \bar{x}_1 \bar{x}_2]$$

where

$$\bar{x}_i = 1 - x_i$$

This can be demonstrated as follows given that the two input sequences are independent.

$$\text{Exp}[x_o] = \text{Exp}[x_1]\text{Exp}[x_2] + \text{cov}(x_1, x_2) + \text{Exp}[\bar{x}_1]\text{Exp}[\bar{x}_1] + \text{cov}(\bar{x}_1, \bar{x}_2)$$

$$\begin{aligned}
\text{cov}(\bar{x}_1, \bar{x}_2) &= \text{Exp}[(1 - x_1)(1 - x_2)] - \text{Exp}[1 - x_1]\text{Exp}[1 - x_2] \\
&= \text{Exp}[1 - x_1 - x_2 + x_1 x_2] - (1 - \text{Exp}[x_1])(1 - \text{Exp}[x_2]) \\
&= \text{Exp}[x_1 x_2] - \text{Exp}[x_1]\text{Exp}[x_2] \\
\Rightarrow \text{cov}(\bar{x}_1, \bar{x}_2) &= \text{cov}(x_1, x_2)
\end{aligned}$$

$$\text{Exp}[x_o] = \text{Exp}[x_1]\text{Exp}[x_2] + \text{Exp}[\bar{x}_1]\text{Exp}[\bar{x}_2] + 2\text{cov}(x_1, x_2)$$

As x_1 and x_2 are independent then

$$\text{cov}(x_1, x_2) = 0$$

thus

$$p_o = p_1 p_2 + (1 - p_1)(1 - p_2)$$

Using the fact that (from eq.(4.8))

$$p_i = \frac{x_i}{2X} + \frac{1}{2}$$

$$x_o = \frac{x_1 x_2}{X}$$

As with the SLU case the output p_o is the normalised product of x_1, x_2 with respect to the range of X is formed.

Assessing the variance of the output of the SLB multiplication eq.(4.10) can be used

$$\text{Var}\left(\frac{\hat{x}_i}{X}\right) = \frac{1 - \left(\frac{x_i}{X}\right)^2}{N}$$

and produce

$$\text{Var}\left(\frac{\hat{x}_o}{X}\right) = \text{Var}\left(\frac{\hat{x}_1 \hat{x}_2}{X^2}\right)$$

This can be demonstrated to be

$$\mathrm{Var}\left(\frac{\hat{x}_o}{X}\right) = \mathrm{Var}\left(\frac{\hat{x}_1}{X}\right) + \mathrm{Var}\left(\frac{\hat{x}_2}{X}\right) - N\mathrm{Var}\left(\frac{\hat{x}_1}{X}\right)\mathrm{Var}\left(\frac{\hat{x}_2}{X}\right)$$

4.4 Addition

In the simplest case for the multiplication of two stochastic pulse streams of the previous section §4.3 ie. SLU signals, a single **AND** gate would suffice. To perform addition of two stochastic pulse streams a corollary might be to use a single **OR** gate. Several problems exist with this suggestion. Firstly, if two probabilities in the range $[0, 1]$ are summed the resultant probability could be greater than unity ie. in the range $[0, 2]$, this is not realisable! Secondly, if an **OR** gate is used and there are two coincident pulses arriving at its inputs only a single pulse will be produced by the gate, a bit of data is lost. Possible solutions to overcome these limitations have been put forward by Gaines [53] and by Leaver [43]. Gaines' main proposal is to perform a weighted sum of inputs, a system which can be used for all linear encoding schemes. Gaines' circuits are reviewed for the three linear strategies followed by Leaver's technique which relies upon insertion of excessive number of pulses into the resulting output stochastic pulse stream. A new appropriate efficient gating circuit is put forward for an N input summer operating upon Gaines' principles.

For the case of the SLU signals, the circuit of Figure 4.7 can be used to perform a weighted sum of two inputs. The two generating probabilities p_1 and p_2 exist for the inputs x_1 and x_2, a third unipolar line S generating probability p_3 acts as a gating signal to determine which of x_1 or x_2 should be switched to the output x_o. A strong resemblance can be seen between Figure 4.6 and Figure 4.7 from which it can be deduced that

$$p_o = p_3 p_1 + (1 - p_3)p_2 \tag{4.16}$$

Using eq.(4.3) the variance of the output can be verified to be

$$\mathrm{Var}(\hat{p}_o) = p_3 \mathrm{Var}(\hat{p}_1) + (1 - p_3)\mathrm{Var}(\hat{p}_2) + (p_1 - p_2)^2 \mathrm{Var}(\hat{p}_3) \tag{4.17}$$

from

$$\mathrm{Var}(\hat{p}_o) = \frac{(p_1p_3 + (1 - p_3)p_2)(1 - (p_1p_3 + (1 - p_3)p_2))}{N}$$

The output of this circuit is

$$x_o = p_3 x_1 + (1 - p_3)x_2 \tag{4.18}$$

If $p_3 = 0.5$ then

$$x_o = \frac{x_1 + x_2}{2}$$

The DLB case is slightly more complex. An initial system would be to use two circuits of Figure 4.7 one for the UP lines and one for the DOWN lines. Thus using eq.(4.16)

$$u_o = p_3 u_1 + (1 - p_3) u_2$$
$$d_o = p_3 d_1 + (1 - p_3) d_2$$
$$\Rightarrow x_o = u_o - d_o = p_3(u_1 - d_1) + (1 - p_3)(u_2 - d_2)$$

By substituting the respective values of u_o and d_o into eq.(4.17) the variance for the result is

$$\mathrm{Var}(\hat{p}_o) = p_3 \mathrm{Var}(\hat{p}_1) + (1 - p_3)\mathrm{Var}(\hat{p}_2) + (p_1 - p_2)^2 \mathrm{Var}(\hat{p}_3)$$

From the above equation it can be seen that if x_1 and x_2 are in a minimum variance form then x_o will not necessarily be in a minimum variance form. This can be explained by the following example, if $(u_1,\ d_2)$ and $(u_2,\ d_1)$ are non-zero ie. the two quantities are of opposite sign, then $(u_o,\ d_o)$ will both be non-zero and the result is not in minimum variance form.

Another circuit approach is that of Figure 4.8 from which it is possible to produce the sum of two inputs in a minimum variance form. This circuit cancels the positive signals on one set of inputs with negative signals upon the other input set.

$$u_o = p_3(1 - d_2)u_1 + (1 - p_3)(1 - d_1)u_2$$
$$d_o = p_3(1 - u_2)d_1 + (1 - p_3)(1 - u_1)d_2$$
$$\Rightarrow u_o - d_o = p_3(u_1 - d_1) + (1 - p_3)(u_2 - d_2) + (1 - 2p_3)(u_1 d_2 - u_2 d_1)$$

which in the case of $p_3 = 0.5$

$$u_o - d_o = \frac{u_1 - d_1}{2} + \frac{u_2 - d_2}{2}$$

From eq.(4.5)

$$x_o = \frac{x_1 + x_2}{2}$$

If u_o and d_o are summed from the above equations

$$u_o + d_o = \frac{u_1 + d_1}{2} + \frac{u_2 + d_2}{2} - u_1 d_2 - u_2 d_1$$

the values of $u_o - d_o$ and $u_o + d_o$ may be substituted into eq.(4.6). The resulting variance value is

$$\mathrm{Var}\left(\frac{\hat{x}_o}{X}\right) = \frac{\left(\frac{\hat{x}_1}{X}\right)}{2} + \frac{\left(\frac{\hat{x}_2}{X}\right)}{2} + \frac{\left(\frac{x_1 - x_2}{X}\right)^2}{4N} - \frac{(u_1 d_2 - u_2 d_1)}{N}$$

The final linear coding scheme of the SLB case is similar to the SLU addition case.

The circuit of Figure 4.7 will suffice again with the result of

$$x_o = p_3 x_1 + (1 - p_3)x_2$$

This time it is generated via the substitution of the generic version of eq.(4.7) into eq.(4.16). Once again if $p_3 = 0.5$ the result

$$x_o = \frac{x_1 + x_2}{2}$$

is arrived at.

This weighted summation format is not the only approach to stochastic addition. Leaver [43] puts forward an alternative strategy that of pulse insertion. One of the problems stated above with using an **OR** gate for the purpose of addition is the gate's failure to account for the condition of coincident pulses upon its inputs. Rather than weight each input pulse train they are both added together using an **OR** gate with any coincident pulses detected by an additional **AND** gate. The output of this **AND** gate is used to increment a counter which holds a record of outstanding coincident pulses. If no pulses are detected as being emitted by the adding **OR** gate and the coincident pulse counter holds a value greater than zero a pulse is generated, inserted back into the output pulse train and the counter decremented. Figure 4.9 shows a circuit which can perform the coincident pulse detection and insertion. For the SLU addition only a single circuit is required, but for DLB addition it is necessary to use one for the UP lines and one for the DOWN lines. In the SLB case a system which detects and accounts for both coincident spaces as well as coincident pulses is required. If there is an excess of pulse pairs then additional pulses must be inserted into the output sequence and if there is an excess of space pairs pulses should be removed from the output sequence.

For all instances of Leaver's adders [43] no scaling of either inputs or output occurs and the output probability can try to exceed the range $[0, 1]$ producing an incorrect addition. Using the SLU system as an example, before the out of bounds condition occurs the probability of coincident pulse pairs will increase requiring a large counter to maintain a record of how many pulses must be inserted. With the output sequence becoming increasingly full as the limit of the adder is approached so a lag may build up for the insertion of pulses back into the output sequence when the input sequences change. This lag will be particularly acute if the result of the summation would be greater than a probability of 1. It is possible to pre-scale the input values into a Leaver adder so as not to exceed the dynamic range, but if this is going to be performed then the extra complexity of using the counter does not appear worthwhile.

4.4.1 An N Input Adder Proposal

In this section a new circuit for the addition of N input signals is proposed since Gaines [53] makes only a passing reference to the problem of the accurate summation of more than two stochastic signals. The simple cascading of summation circuits presented so far will not suffice in the general case. For Leaver's adders the result is more likely to tend towards a limiting factor of a saturated pulse stream so reducing the accuracy of the addition or the magnitude of signals which could be summed unless pre-scaling the inputs occurs. Using Gaines's two input weighted summer the result for three sequences would be

$$x_o = \frac{\frac{x_1+x_2}{2} + x_3}{2} = \frac{x_1}{4} + \frac{x_2}{4} + \frac{x_3}{2}$$

However, it can be seen that if the number of sequences to be added is a power of two such a system would succeed. This may not be practical for a particular application. What is desired is, for in the case of three lines, three weighting sequences of value $\frac{1}{3}$ with no two weighting sequences having coincident pulses giving the result of eq.(4.19).

$$x_o = \frac{x_1 + x_2 + x_3}{3} \tag{4.19}$$

For the general case of adding N sequences it is necessary to weight each of the sequences by $\frac{1}{N}$ ensuring that all pulses in each sequence are mutually exclusive. This last condition will mean that the weighting sequences are not statistically independent.

Let us assume that the summation of N pulse streams is desired. First a unipolar sequence of $\frac{1}{N}$ is generated, the first $\frac{1}{N}$ sequence. Complementing this sequence using an inverter will generate $(1 - \frac{1}{N}) \equiv \left(\frac{N-1}{N}\right)$. A new sequence of $\frac{1}{N-1}$ is generated which by taking the product of $\left(\frac{1}{N-1}\right)\left(\frac{N-1}{N}\right)$ forms a second $\frac{1}{N}$ sequence. This process is continued with the sequence $\frac{1}{N-2}$ generated and multiplied by the complements of both $\frac{1}{N}$ and $\frac{1}{N-1}$ to form another $\frac{1}{N}$. This process is fully illustrated by Table 4.1.

Pulse Sequence of N	Weighting Calculation	Output Weight
1	$\frac{1}{N}$	$\frac{1}{N}$
2	$\frac{1}{N-1}\left(1 - \frac{1}{N}\right)$	$\frac{1}{N}$
$\vdots$	$\vdots$	$\vdots$
$N-1$	$\frac{1}{2}\left(1 - \frac{1}{3}\right)\cdots\left(1 - \frac{1}{N-1}\right)\left(1 - \frac{1}{N}\right)$	$\frac{1}{N}$
N	$\frac{1}{1}\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right)\cdots\left(1 - \frac{1}{N-1}\right)\left(1 - \frac{1}{N}\right)$	$\frac{1}{N}$

Table 4.1: Weighting calculations for N pulse sequences of value $\frac{1}{N}$

This process of forming N $\frac{1}{N}$ sequences is effective because the complement of a pulse sequence has no coincident pulses with its original. The product of complements will thus have no coincident pulse with any of the generating sequences thus by multiplying by the next $\frac{1}{N-z}$ will produce a new suitable $\frac{1}{N}$ pulse sequence.

What actual form should the base $\frac{1}{N}$, $\frac{1}{N-1}$, $\ldots$, $\frac{1}{2}$ sequence take? It can be demonstrated graphically that deterministic pulse sequences would need to be judiciously selected or else unsatisfactory results are produced. Figure 4.10 illustrates clearly the problem with deterministic sequences for four $\frac{1}{4}$ signals. The mathematical operations of multiplication and addition dealt with so far have been conducted in the stochastic domain. Using stochastic pulse sequences for this divider does produce the desired response. A short piece of computer code can be produced to demonstrate this principle operating effectively.

Finally, this stochastic N pulse stream weights must be sensibly realised in hardware. It can be seen from the equations describing the weight functions that a cascade of complementer (inverters) and multipliers (**AND** gates) is all that is required, Figure 4.11. Two problems are immediately apparent from the schematic of Figure 4.11 as follows,

1. the loading upon the inverters at the top of the cascade will be detrimental to the performance.

2. the required fan-in of the **AND** gates at the bottom of the cascade will be large.

The greater the number of sequences the more acute the two problems will become. Due to the repetitive and modular nature of the expansion to create the sequences the circuit of Figure 4.11 can be improved upon to Figure 4.12. Figure 4.12 takes advantage of the repetitiveness with an improved circuit design. No undue loading is placed upon the inverters at the top of the cascade and the fan-in of all the **AND** gates remains at two regardless of their position in the cascade. This second design is not without its drawbacks, the greater the value of N the greater the propagation delay for the pulses to ripple down the cascade, resulting in the output pulses not being synchronised and spikes forming by partial results. Despite this, N $\frac{1}{N}$ pulse sequences can be adequately generated and used to weight the input to an **OR** gate adder.

4.5 Subtraction

The subtraction operation only really needs to be considered for unipolar signals. For bipolar signals subtraction is achieved by the addition of negative or complements of the desired signals.

4.5.1 A Subtracter Proposal

For unipolar signals a negative signal representation does not exist, but translating Leaver's technique of pulse insertion for addition to one of pulse removal for subtraction the desired operation can be effected. Figure 4.13 illustrates schematically a circuit proposal to achieve subtraction for unipolar signals. For this circuit, pulse stream y is being subtracted from pulse stream x. Pulses on y are accumulated in a counter the output of which is active if the counter's content is greater than zero. The **AND** gate will produce an output with the next pulse upon x which is removed from the output by means of the **XOR** gate. The output from the **AND** gate also decrements the counter, since one less pulse has to be removed from x.

Problems with this circuit will occur if the number of pulses in y is greater than those in x for a sustained period of time, $y > x$. In effect an attempt will be made to exceed the lower probability bound of zero. The counter will count up thus when y is less than x again and a valid subtraction can be performed a lag results as the counter removes pulses and decrements before settling down to produce a correct result. Although this system is not ideal and no account is taken as to whether a negative result would be the outcome it does demonstrate that subtraction could be achieved. In general it is required that $x > y$ for valid subtractions to be performed.

4.6 Integration and the ADDIE

The preceding sections of this chapter have discussed computations which use only combinational logic elements and have no knowledge of the previous events. More sophisticated operations eg. square-rooting and function generation, may be formulated using integrators. Integration requires knowledge of previous events and thus memory is required. Integration is the summing of preceding events which can be accomplished by use of a digital up/down counter. The counter increments by one if the UP line is active on a clock pulse, decrements by one if the DOWN line is active on a clock pulse and remains unaltered if both lines are in the same state, assuming a DLB system.

The counter can be considered to have $N+1$ states, $S = S_0, S_1, \cdots, S_N$ where s_i is the numerical value of each state and also the output of the counter when it is the i'th state. A possible linear mapping from the value held in the counter into the range $(0, 1)$ is

$$s_i = \frac{i}{N}$$

At a given time the counter is in a state $S = S_i$ with output $s = s_i$. Driving the counter with stochastic sequences means that the actual counter state is unpredictable but it may be expressed as a probability π_i. The output is now a random variable with expected value $\bar{s}$ defined as

$$\bar{s} = \sum_{i=0}^{N} \pi_i s_i$$

Using a Bernoulli sequence to drive the UP and DOWN lines of the counter, such that the probability of the UP line being on and the DOWN line being off is w, and the probability that the UP line is OFF and the DOWN line is on is e, the expected change of the counter output is

$$\delta s = \frac{w - e}{N}$$

Over a clock period T seconds the expected counter output change is

$$\bar{s}(n\mathrm{T}) - \bar{\mathrm{s}}(0) = \sum_{\mathrm{n}=0}^{\mathrm{m}-1} \delta \mathrm{s}(\mathrm{nT}) = \sum_{\mathrm{n}=0}^{\mathrm{m}-1} \frac{\mathrm{w(nT)} - \mathrm{e(nT)}}{\mathrm{N}} \tag{4.20}$$

eq.(4.20) is a simple zero-order numerical integration formula for $w(t) - e(t)$ which can be reorganised and rewritten as

$$\bar{s}(t) = s(0) + \frac{1}{N\mathrm{T}} \int_0^{\mathrm{T}} w(\tau) - e(\tau) d\tau$$

SLU, DLB and SLB mappings can be used to implement this integration technique with a counter as will now be considered.

Only positive quantities exist for SLU signals and the counter can only count up. The data line is connected to the up port of the counter with the down port set to off. The quantity being integrated is x_1, the quantity represented by the counter is x_o,

$$w = \frac{x_1}{X}$$
$$e = 0$$
$$s = \frac{x_o}{X}$$
$$x_o(t) = x_o(0) + \frac{1}{N\mathrm{T}} \int_0^t x_1(\tau) d\tau$$

In the DLB representation the UP and DOWN lines for the signal can be connected directly to the up and down ports of the counter respectively. A transformation mapping is now appropriate for the output of the counter since bipolar quantities are represented.

$$\frac{x_o}{X} = (2\bar{s} - 1)$$

that the UP line is OFF and the DOWN line is on is e, the expected change of the counter output is

$$\delta s = \frac{w - e}{N}$$

Over a clock period T seconds the expected counter output change is

$$\bar{s}(n\mathrm{T}) - \bar{\mathrm{s}}(0) = \sum_{\mathrm{n}=0}^{\mathrm{m}-1} \delta \mathrm{s}(\mathrm{nT}) = \sum_{\mathrm{n}=0}^{\mathrm{m}-1} \frac{\mathrm{w}(\mathrm{nT}) - \mathrm{e}(\mathrm{nT})}{\mathrm{N}} \tag{4.20}$$

eq.(4.20) is a simple zero-order numerical integration formula for $w(t) - e(t)$ which can be reorganised and rewritten as

$$\bar{s}(t) = s(0) + \frac{1}{N\mathrm{T}} \int_0^{\mathrm{T}} w(\tau) - e(\tau) d\tau$$

SLU, DLB and SLB mappings can be used to implement this integration technique with a counter as will now be considered.

Only positive quantities exist for SLU signals and the counter can only count up. The data line is connected to the up port of the counter with the down port set to off. The quantity being integrated is x_1, the quantity represented by the counter is x_o,

$$w = \frac{x_1}{X}$$

$$e = 0$$

$$s = \frac{x_o}{X}$$

$$x_o(t) = x_o(0) + \frac{1}{N\mathrm{T}} \int_0^t x_1(\tau) d\tau$$

In the DLB representation the UP and DOWN lines for the signal can be connected directly to the up and down ports of the counter respectively. A transformation mapping is now appropriate for the output of the counter since bipolar quantities are represented.

$$\frac{x_o}{X} = (2\bar{s} - 1)$$

Let x_1 be the value on the input lines with the following probabilities defined, then

$$w = u_1$$

$$e = d_1$$

$$\frac{x_1}{X} = u_1 - d_1 = w - e$$

$$\Rightarrow x_o(t) = x_o(0) + \frac{2}{N\mathrm{T}} \int_0^t x_1 \tau d\tau$$

Due to the transformation mapping the effective gain of the integrator has increased by a factor of two.

For the final encoding scheme of SLB the integrator is formed by connecting the signal line directly to the up port of the counter and connecting an inverted form to the down port. The quantity x_1 is represented by the generating probability p_1, therefore

$$w = p_1$$

$$e = 1 - p_1$$

and

$$w - e = 2p_1 - 1 = \frac{x_1}{X}$$

$$\Rightarrow x_o(t) = x_o(0) + \frac{2}{N\mathrm{T}} \int_0^t x_1 \tau d\tau$$

The next advance from these single input integrators is to dual input integrators. Quite obviously it is feasible to precede the single input integrator with a two input addition circuit from §4.4, but for the bipolar systems a saving in hardware can be gained by judicious gating prior to the counter to form a two input summing integrator. A slightly more sophisticated counter is required in the case of the dual line representation.

Using the circuit of Figure 4.14 for DLB signals, which necessitates a counter which can increment and decrement by two, an equally weighted integration can be performed. If the UP2 line is on when both UP_1 and UP_2 lines are on then the counter increments by two. The UP1 line is on if only one UP line is on and similarly for the down lines.

If the UP and DOWN lines of each input are subscripted 1 and 2 respectively then the expected change in output s, δs, is given by

$$\delta s = \frac{2u_1u_2 + u_1(1 - u_2 - d_2) + u_2(1 - u_1 - d_1) - d_1(1 - u_2 - d_2) - d_2(1 - u_1 - d_1) - 2d_1d_2}{N}$$

$$\delta s = \frac{u_1 - d_1 + u_2 - d_2}{N} = \frac{w - e}{N}$$

$$x_o(t) = x_o(t) + \frac{1}{N\mathrm{T}} \int_0^t (x_1(\tau) + x_2(\tau) d\tau$$

For the SLB case, the integration of the sum of two inputs is achieved by utilisation of the three possible input conditions. The counter increments if both lines indicate up, the counter decrements if both lines indicate down and no change occurs if the two inputs are opposite. Figure 4.15 shows the required gating,

$$w = p_1 p_2$$

$$e = (1 - p_1)(1 - p_2)$$

$$w - e = p_1 + p_2 - 1 = \frac{(2p_1 - 1) + (2p_2 - 1)}{2}$$

$$w - e = \frac{x_1 + x_2}{2X}$$

$$x_o(t) = x_o(t) + \frac{1}{NT} \int_0^t (x_1(\tau) + x_2(\tau) d\tau$$

The output of all the integrator circuits discussed have been a state S_i with a value s_i which can be read out from the counter as either a parallel or serial bit values. This value is no longer within the stochastic pulse domain. To continue pulse processing it is necessary to re-encode the value s_i back into the stochastic pulse domain. Re-encoding is achieved as with the basic encoding strategies of §4.1 dependant upon the representation scheme adopted. The integrator can be summarised as Figure 4.16.

The ADDIE, Adaptive Digital Element, is formed from a two input summing integrator. Its operation depends upon the stochastic input sequence and the probability of the feedback sequence from the current state of the integrator, Figure 4.17. The ADDIE is used as the basis for output interfaces discussed in a following section, §4.8. The operation of the ADDIE can be explained by reference to a passive frequency modulation detector, [77]. The input to the circuit of Figure 4.18 is a fixed frequency train of pulses. A steady state voltage will be output depending upon the frequency of the incoming sequence when the rate of charging by the pulses is balanced by the discharge rate through the resistor. The ideal case will be that the voltage across the capacitor will be directly proportional to the rate of discharge.

$$v = \mathrm{K}\frac{dv}{dt}$$

$$\log v = -\frac{1}{\mathrm{K}}t + c$$

at $t = 0$

$$v = V_0 e^{-\frac{t}{\mathrm{K}}} \tag{4.21}$$

The RC network realises eq.(4.21). Moving forward to a pulse train which has a varying frequency but that the frequency is around a fixed mean value, the voltage across the capacitor will vary but with a fixed mean value. Advancing again to the analogue circuit representation of this frequency detector, Figure 4.19, the output voltage is now dependant upon the ratio of the two resistors, $\frac{R_1}{R_2}$. For a circuit with purely capacitive feedback, integration is performed equivalent to that of the up/down counter of the stochastic circuits. The negative feedback resistor is equivalent to the inverted output fed back in the stochastic circuit, Figure 4.17. The ADDIE operating upon stochastic pulse sequences thus has similar characteristics to the RC network upon an operational amplifier. The state of the counter is a binary number proportional to the probability of the input stochastic sequence. The value of the ADDIE time constant is varied by adjusting the counter length or applying a multiplier to the feedback stochastic pulse train.

The ADDIE may be used as the basis for function formation as described by Gaines [53]. For example, the square root of a number may be extracted by feeding back the square of the inverse of the ADDIE output rather than simply the inverse, Figure 4.20. Note, in this circuit, the D-type flip-flop delays the fed back pulse stream by one cycle effectively isolating the pulse stream from itself and making it statistically independent, hence enabling squares at the multiplier to occur.

The functionality of the ADDIE may be further extended by connecting a gating circuit to the ADDIE's counter. The integrator's counter will contain an increasingly accurate estimate of the probability that the input line is on. Thus, the counter gating may be used to apply arbitrary transformations to the stored count. The transformed quantity can be re-encoded into a stochastic pulse sequence for further processing. Figure 4.21 illustrates the configuration for such a system.

4.7 Sigmoidal Transform Proposal

It is aimed to produce a sigmoidal transfer function for use in a neuron design operating using stochastic pulse sequences. It is desired to keep all operations digital and within the stochastic processing domain. Several options can be considered for forming this sigmoidal transfer function, forming the sigmoid function equation stochastically using the ADDIEs, implementing a look-up table of input to output values and finally a non-linear stochastic transform. Each of these three will be considered in turn.

Using ADDIEs to formulate the sigmoid function equations would require one of the following equations to be produced,

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4.22}$$

or

$$f(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{4.23}$$

Directly realising the exponential function is not feasible using stochastic circuits, but eq.(4.22) and eq.(4.23) could be represented by a power series using a Maclaurin's expansion. For eq.(4.22) this produces

$$f(x) = \frac{1}{1+e^{-x}} \approx \frac{1}{2} + \frac{1}{4}x - \frac{1}{48}x^3 + \cdots$$

and for eq.(4.23) produces,

$$f(x) = \frac{1-e^{-2x}}{1+e^{-2x}} \approx x - \frac{1}{3}x^3 + \frac{2}{15}x^5 + \cdots$$

NB. These are not the only sigmoidal equations but they are the ones most commonly used.

The accuracy of these expansions is limited. The scaling terms could be formed in a similar manner to that used in the $N+1$ pulse divider but would require large pulse divider circuits. Therefore, this method for forming a sigmoid from a base equation is not recommended.

Using a look-up table requires that the input values to the table from the output value formed from an ADDIE are a stable quantity. This quantity is used to reference a corresponding value which is encoded into the stochastic pulse domain. The profile and accuracy of the sigmoid formed will depend upon the number of elements in the table and thus the length of the counter in the ADDIE. Using a look-up table requires the transfer out from and back into the stochastic pulse domain. This is an entirely digital system.

The third option is a non linear stochastic transform which will now be demonstrated in the following sections. This transform utilises Even-Shift orthogonal sequences which can used to form a Gaussian random number (GRN) generator. This GRN is used to perform the actual transform by comparison with a stochastic pulse sequence. A circuit is presented to actually carry out the transfer function.

4.7.1 Even-Shift Orthogonal Sequences

An Even-Shift Orthogonal Sequence, *E-sequence*, is defined as a sequence of length n, $S = (s_1, s_2, \ldots, s_n)$ whose elements s_j $(j = 1, 2, \ldots, n)$ are either 1 or -1 and whose auto-correlation function $\Psi_{ss}(i)$ is zero for all even shifts except the zero shift, [78].

$$\Psi_{ss}(i) = \sum_{k=1}^{n-|i|} s_k s_{k+|i|} = 0 \tag{4.24}$$

$$i = \pm 2, \pm 4, \ldots \pm (n-2)$$

Figure 4.22 illustrates the auto-correlation function for the following 16 element E-sequence, (-1, 1, -1, 1, 1, 1, -1, -1, -1, 1, 1, -1, 1, 1, 1, 1). E-sequences are derived from and have a one-to-one correspondence with complementary sequences discussed by Golay, [79]. As such it can be shown that the length n of an E-sequence is an integral multiple of four and that n must be twice the sum of at most two square numbers. These are not apparently sufficient conditions though.

Given an E-sequence, S, as defined above, the sequence can be decomposed into the form

$$S \equiv (X;Y) \tag{4.25}$$

where

$$Xr = (s_1, s_3, \ldots, s_{n-1})$$

$$Yr = (s_2, s_4, \ldots, s_n)$$

X expresses the sequence of odd-number subscripted elements, while Y is the even-number subscripted elements of S. These two sequences X and Y, form a pair of complementary sequences of length $\frac{n}{2}$. Thus, given a pair of complementary sequences X and Y, the binary sequence formed by eq.(4.25) is an E-sequence.

It can be demonstrated and verified that for an E-sequence $(X;Y)$ the following combinations are also E- sequences: $(Y;X)$, $(X^R;Y)$, $(X;Y^R)$, $(X^R;Y^R)$, $(-X;Y)$, $(X;-Y)$, $(-X;-Y)$, $(X_{A_o};y_{A_o})$, $(X_{A_o};Y_{A_e})$, $(X_{A_e};Y_{A_o})$, $(X_{A_e};Y_{A_e})$. The superscript R stands for reversing the order of the elements. The subscripts A_o and A_e stand for inverting the sign of the odd or even elements of the subsequence respectively.

Although methods exist for forming one E-sequence from another E-sequence and from complementary pair sequences, no reference could be found for a method determining the number of E-sequences of a given length or calculating them all, other than by an exhaustive search through all sequences to find those which satisfy eq.(4.24). Software was written using Borland Turbo C++ version 2 to test all possible sequences. A problem immediately becomes apparent with this search; as the number of bits for prospective E-sequences increase by one, the search space doubles. The runtime of the program increases exponentially with n.

4.7.2 Sigmoidal Transform Production Using Gaussian Distributed Random Numbers

In §4.1 encoding or input mapping techniques have been discussed using uniform distributed random numbers to map a deterministic value into a stochastic pulse stream of 1's and 0's. In general, the probability of observing a one on the output line represents the normalised deterministic value. These linear transfer functions are the Cumulative Distribution Function (CDF) for a uniform random number which has a Probability Density Function (PDF) as illustrated in Figure 4.23.

If we require a sigmoidal transfer function, ie. CDF as Van Den Bout [56] explains, it is necessary to find an appropriate PDF to encode the variable against. The *Gaussian* or

Normal distribution function has the following PDF eq.(4.26)

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad -\infty < x < \infty \tag{4.26}$$

where μ is the mean value of the distribution and σ^2 is the variance. The associated CDF is eq.(4.27)

$$F(x) = \int_{-\infty}^{a} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \tag{4.27}$$

For $\mu = 0$ and $\sigma^2 = 1$ Figure 4.24 shows the respective graphs.

It can be seen from eq.(4.26) and eq.(4.27) that the offset of the PDF and therefore the CDF is governed by the mean value of the Gaussian distribution. The variance of the distribution affects the *peakiness* of the PDF which in turn affects the sharpness of the sigmoidal transform of the CDF. The results of adjusting the mean and variance upon the CDF output are illustrated in Figure 4.25 and Figure 4.26.

Increasing the variance reduces the gradient of the sigmoid, decreasing the variance increases the gradient. Increasing the mean moves the sigmoid to the right, while decreasing the mean moves the sigmoid, in the opposite direction, to the left. Thus by manipulation of the variance and mean the resulting sigmoid can be altered. These two sets of results were plotted from the output generated by a simple software model.

4.7.3 Sigmoidal Transform Production Using E-Sequences

It is well known that a Gaussian random signal may be generated via the *Central Limit Theorem*[2]. Broadly the central limit theorem states that the sum of n identically generated independent random variables tends towards a Gaussian distribution as $n \longrightarrow \infty$. An approximation can be realised by the addition of n binary random variables with a digital filter which has a weighting function of n weight elements.

Izumi [81] proposes the use of an E-sequence for the digital filter weighting function based upon the ideas of Davies [82] and his own earlier work [83]. An E-sequence weighting function is selected since it is an optimum weighting function for the production of a Gaussian distribution. The quality of the produced Gaussian distribution is measured in

[2]**Central Limit Theorem** [80]
Let $X_1, \ldots, X_n$ be independent random variables that have the same distribution function and therefore the same mean μ and the same variance σ^2. Let $Y_n = X_1 + \ldots + X_n$, then the random variable

$$Z_n = \frac{Y_n - n\mu}{\sigma\sqrt{n}}$$

is **asymptotically normal** with mean 0 and variance 1; ie. the distribution function $F_n(x)$ of Z_n satisfies

$$\lim_{n\to\infty} F_n(x) = \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-u^2}{2}} du$$

terms of the coefficient of *skewness*[3] and the coefficient of *kurtosis*.[4] Izumi subsequently demonstrates the suitability of an *E*-sequence.

Developing the circuit used by Izumi to create a Gaussian random number the desired sigmoidal transform can be formed by using the Gaussian random number to map a value into a stochastic pulse stream. A block diagram of the proposed circuit is shown in Figure 4.27.

Pulses from a Pseudo Random Binary Sequence (PRBS) are weighted by the values of an *E*-sequence. The resultant products are accumulated in an Up/Down Counter which has been pre-loaded with the offset for the Gaussian mean. After the entire *E*-sequence has been cycled through, n products, the value of the counter is output to a comparator to map the required value x into the probability of a pulse according to a sigmoidal transform. If the number of bits for x is more than produced by the counter, the output of the counter has zeros padded for the least significant bits.

Binary values are being manipulated so the *E*-sequence is represented in terms of 1's and 0's as opposed to 1's and -1's. The derivation for the Increment and Decrement signals is shown in Table 4.2.

PRBS Bit r_i	*E*-Sequence Bit w_i		$r_i w_i$	Increment	Decrement
	Bipolar	Binary	Bipolar	Binary	Binary
0	-1	0	0	0	0
0	1	1	0	0	0
1	-1	0	-1	0	1
1	1	1	1	1	0

Table 4.2: Derivation of Increment and Decrement Gating for Gaussian Random Number Generator

What does the sigmoid look like which is produced by this circuit? What effect does the zero padding have? To investigate these two areas, a simple software model was written. This produces results of the input/output relationship for the sigmoidal transform. Figure 4.28 illustrates a typical sigmoid formed for a given *E*-sequence. In fact, all sigmoids were found to have this appearance regardless of the number of zeros used for LSB padding provided the input encoding range had a similar number of bits. This is due to

[3]**Coefficient of Skewness** [84] is the 3rd moment of X^* and is denoted by γ_1.

$$\gamma_1 = \mathrm{E}(X^{*3}) = \sigma^{-3}\mathrm{E}\{(X-\mu)^3\}$$

If the distribution of X is symmetrical about μ eg. uniform distribution, binomial distribution then $\gamma_1 = 0$. If X has a long tail to the right, eg. geometric distribution, Poisson distribution, $\gamma_1 > 0$ the distribution is said to be positively skewed. If X has a long tail to the left $\gamma_1 < 0$ and the distribution is negatively skewed.

[4]**Coefficient of Kurtosis** [84] is 3 less than the 4th moment of X^* and is denoted by γ_2.

$$\gamma_2 = \mathrm{E}(X^{*4}) - 3 = \sigma^{-4}\mathrm{E}\{(X-\mu)^4\} - 3$$

The 4th moment is decreased by 3 so that a Gaussian distribution has $\gamma_2 = 0$. A distribution with thicker tails than the Gaussian distribution will have $\gamma_2 > 0$, while one with thinner tails will have $\gamma_2 < 0$.

the fact that the relative dynamic range of the Gaussian random number will be the same. It should also be noted that the sigmoid transform produced is very subtle, but it does exist.

A Gaussian distributed random number with a greater dynamic range is necessary to produce a better quality sigmoidal transform. For a greater dynamic range a larger E-sequence is required which will reduce the frequency with which a Gaussian random number can be generated from a PRBS. The size of the shift register to hold the E-sequence will also increase possibly leading to problems of hardware realisation, but nothing which can not be accommodated.

4.7.4 E-Sequence Conclusions

Following a very brief summary of the properties of E-sequences relevant to their formation and application to the production of sigmoidal transforms, the transformations possible when moving from a PDF to a CDF for a *random number* are discussed with particular reference to Gaussian distributed random numbers. The effects of adjusting the mean and variance for a Gaussian random number upon the transformation are demonstrated. Finally, a circuit for producing a sigmoidal transformation entirely digitally in the stochastic pulse rate encoded domain is proposed.

The sigmoidal transformation circuit proposed has several limitations which include the need for a long shift register to hold the E-sequence, limited dynamic range of the Gaussian random number produced and poor resulting sigmoidal transform. Yet, a sigmoidal transform is produced. Due to the length of the E-sequence a Gaussian random number can only be produced every n clock cycles, where n is the length of the E-sequence. By investigating other E-sequences of the same length or longer, more suitable sequences may be found. Software has been produced to find E-sequences of a given length although at present it is serial and slow for E-sequences of length greater than 24 bits.

4.8 Decoding and Output Interfacing

The majority of the elements described so far have consisted of basic logic gates and have been concerned with processing stochastic pulse signals. At some stage it will be necessary to view the results of any computation. The stochastic value must be converted to a deterministic value.

At a basic level the number of ON pulses for a stochastic pulse sequence are summated over a known number of clock cycles. The ratio of ON pulses to the total number of clock cycles represents an estimate of the sequence value. Increasing the number of clock intervals over which the calculation is performed improves the accuracy but also increases the time over which the measurement is made. If the sequence is stationary, a fixed quantity, this does not pose a problem, however, if the signal is time-varying it is necessary to continually track the signal. Therfore, any system to perform this decoding must have the following characteristics, [75].

- Minimum bias error in the steady-state.
- Minimum variance in the steady-state.
- Minimum response time to a minimum bias error for a step input.
- Minimum response time to minimum variance for a step input.
- Ability to track non-stationary input quickly and accurately.

The solution to this problem is normally a form of Moving Average or Exponential calculation.

A moving average can be maintained by keeping a record of the previous N sequence values and calculating the average pulse rate. For the next clock cycle the oldest sequence value is removed and replaced by the new sequence value and the average is recalculated. If the value on the signal line is represented by $A_i(0,1)$ then the estimate of the sequence value is

$$\hat{p}_N = \frac{1}{N}\left(\sum_{i=1}^{N-1} A_i + A_N\right)$$

this can be shown to be

$$\hat{p}_N = \hat{p}_{N-1} + \frac{A_n - A_0}{N}$$

The shorter the sampling period the greater the effect the new sequence value will have, but the quicker the system response. The inverse is true that the longer the sequence the less influence the new value has but the slower the system is to respond, the bandwidth has been reduced. A major problem with this system is the necessity to store the N previous sequence values. An appropriately long shift register can be utilised as illustrated in the practical circuit Figure 4.29 (cf. Figure 4.15) which performs the second form of moving average calculation.

More sophisticated systems for generating an output can be achieved by adjusting the weighting coefficients applied to the pulse sequence, from the uniform value of $\frac{1}{N}$, providing that the sum of the weights is always unity. Using the ADDIE of §4.6 Mars *et al*, [75], fully explain the use of two ADDIE variants. The first is an ordinary noise ADDIE to produce an output which is an exponential average. The second uses a deterministic pulse stream for feedback and is the Binary Rate Multiplier (BRM) ADDIE. The speed of response of an ADDIE for output is related to the number of bits it uses, the more bits the slower the response to changes in input probability. However, the more bits used the greater the accuracy of the exponential average achieved for a stationary signal.

4.9 Summary

The main aim of this chapter has been to provide an overall critical review of stochastic pulse rate computation by the use of three linear encoding schemes SLU, DLB and SLB. In reviewing this material, primarily of Gaines, the mathematics and logic circuits for performing encoding, inversion, multiplication, addition, subtraction, integration, function formation and decoding have been presented.

In the process of this review a system for actually accurately summating N stochastic pulse sequences has been proposed together with an efficient logic circuit implementation, §4.4.1. Leaver's principle of addition by pulse insertion for SLU signals has been considered and a circuit operating in a similar manner put forward for performing subtraction by pulse removal, §4.5.1. The final new material considered is that of developing a suitable circuit to perform sigmoidal transformations. A circuit using GRNs generated from E-sequences is explained and has been simulated, §4.7. The limitations of this approach are slowness of operation, requirement for a long E-sequence for a reasonable dynamic range and limited quality sigmoid produced but nevertheless a non-linear transformation, sigmoid transform, is generated

Stochastic pulse rate computation relies heavily upon the ability to encode information efficiently using many noise source or suitable random number generators. The following chapter, §5, discusses the generation of random numbers with a view to the efficient parallel generation of several random numbers at once for use in a stochastic pulse processing circuit. With all the constituent parts for an artificial neuron considered the design, implementation and test of an artificial neuron operating using stochastic pulse rate encoded signals is described in Chapter 6.

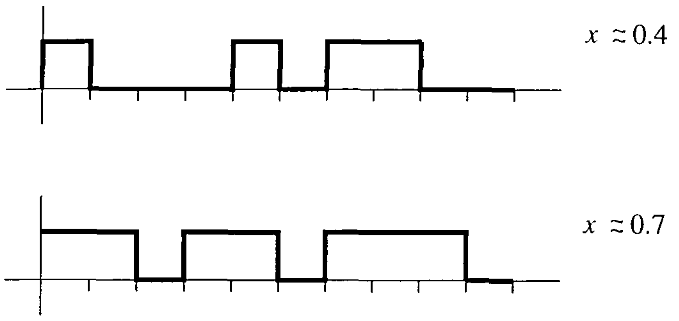


Figure 4.1: Sample encoded pulse streams for an SLU input mapping. *The signal value is the probability of reading a one from the signal line.*

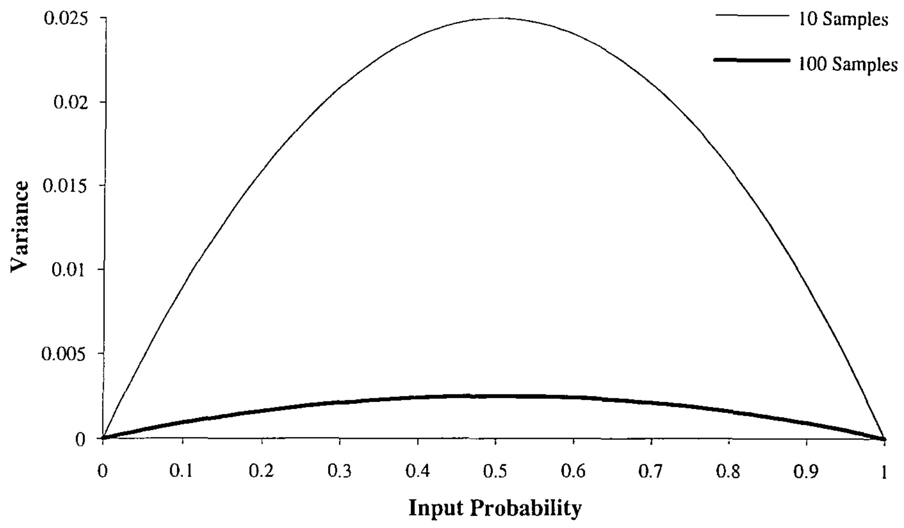


Figure 4.2: Input Probability vs Variance for a SLU Encoding. *This illustrates that the greatest variance occurs at $p = 0.5$ and the balance between speed and accuracy. The more samples obtained the smaller the variance but the longer it will take.*

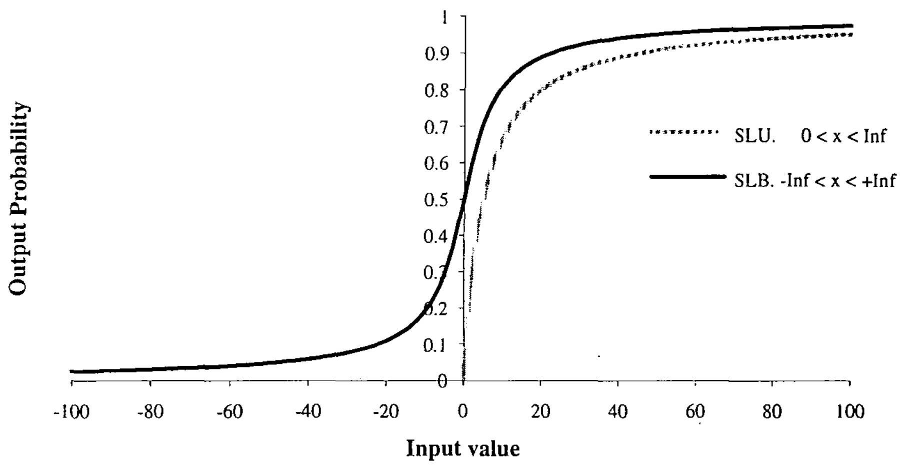
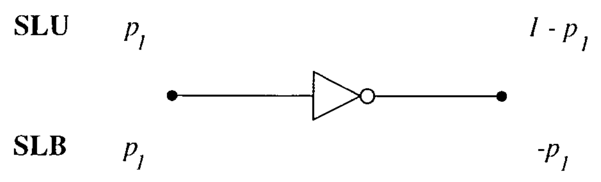


Figure 4.3: Non-linear encoding transfer functions. *Using non-linear encoding systems an infinite range of values can be encoded into the stochastic pulse rate domain.*



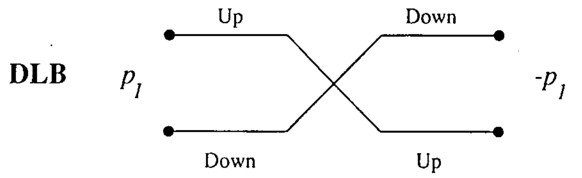


Figure 4.4: Inversion for SLU, SLB and DLB. *A single logical inverter may be used for SLU and SLB signals. Exchanging the two lines is sufficient for DLB signals.*

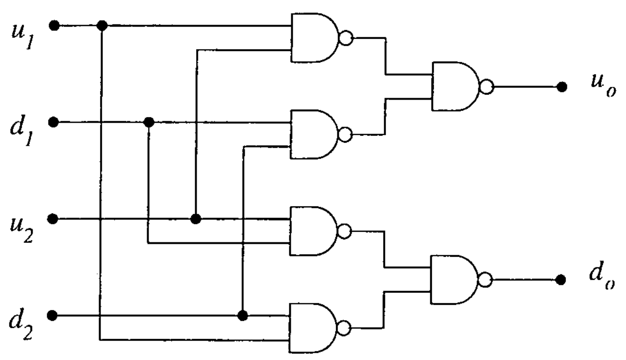


Figure 4.5: DLB multiplication. *If* u_1 *and* u_2 *are high or* d_1 *or* d_2 *are high* u_o *must be high, else* d_o *is high.*

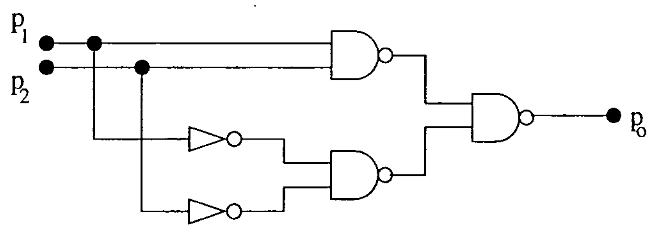


Figure 4.6: SLB multiplication. *An output high is produced if both input signals are in the same state, else an output low is produced.*

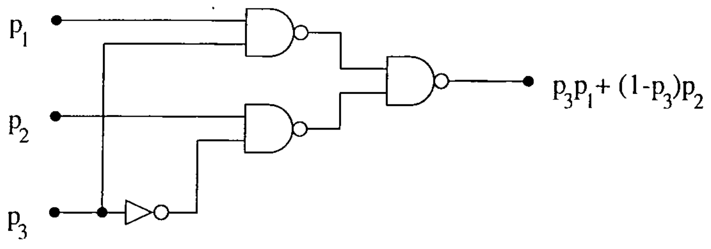


Figure 4.7: SLU/SLB Addition. *The weighted summation of two signals, p_1 and p_2, by a third p_3.*

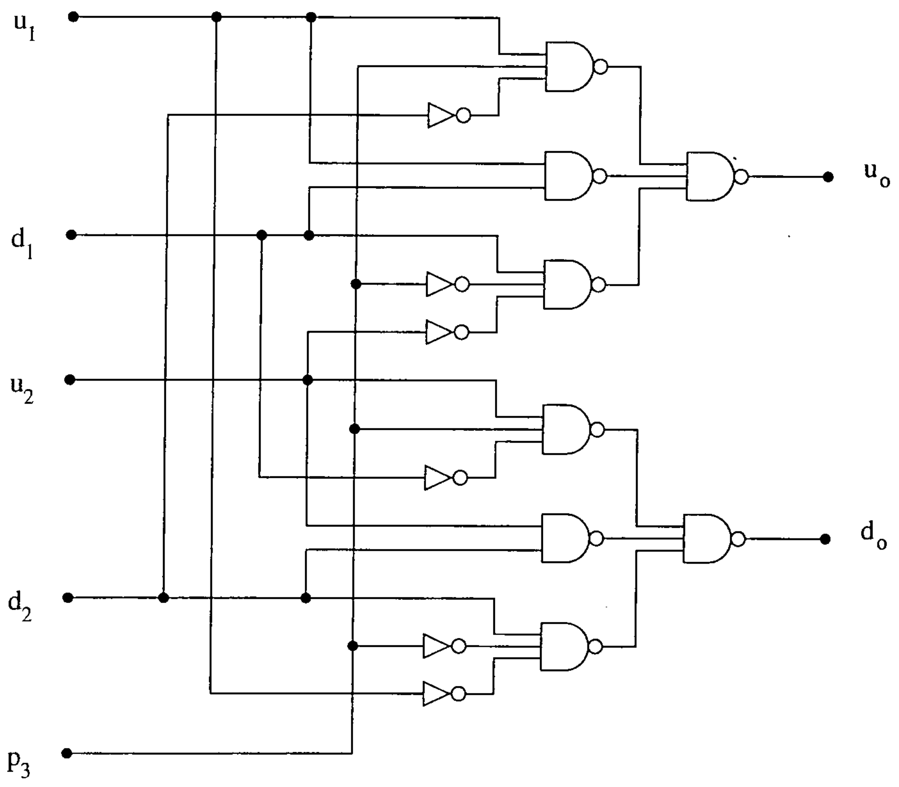


Figure 4.8: DLB addition. *This circuit produces the minimum variance summation of two input signals.*

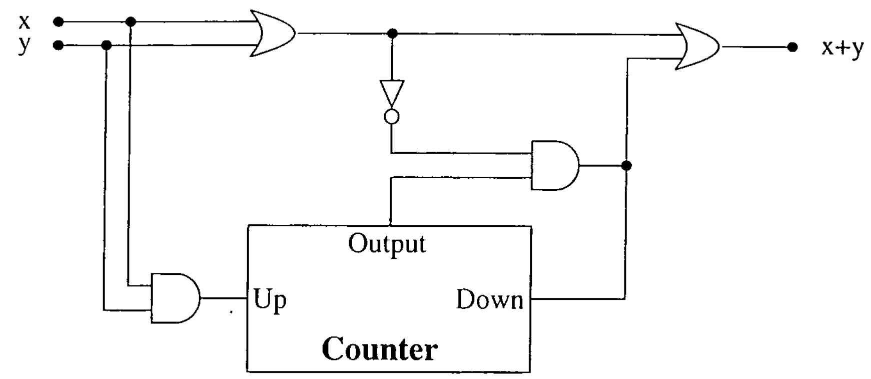
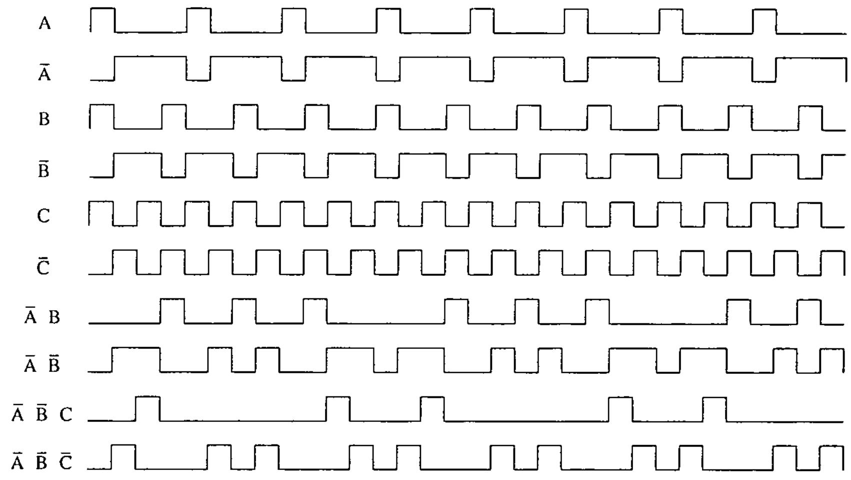


Figure 4.9: SLU addition by pulse insertion. *Coincident pulses upon x and y are accumulated, when both x and y are zero an accumulated pulse is inserted back into the pulse stream.*



A	$\frac{1}{4}$	$\bar{C}$	$\frac{1}{2}$
$\bar{A}$	$\frac{3}{4}$	$\bar{A}\,B$	$\frac{1}{4}$
B	$\frac{1}{3}$	$\bar{A}\,\bar{B}$	$\frac{1}{2}$
$\bar{B}$	$\frac{2}{3}$	$\bar{A}\,\bar{B}\,C$	$\frac{1}{6}$
C	$\frac{1}{2}$	$\bar{A}\,\bar{B}\,\bar{C}$	$\frac{2}{3}$

Figure 4.10: Deterministic sequences for addition. *This diagram demonstrates that deterministic selection of scaling signals can lead to an unequal distribution of pulses.*

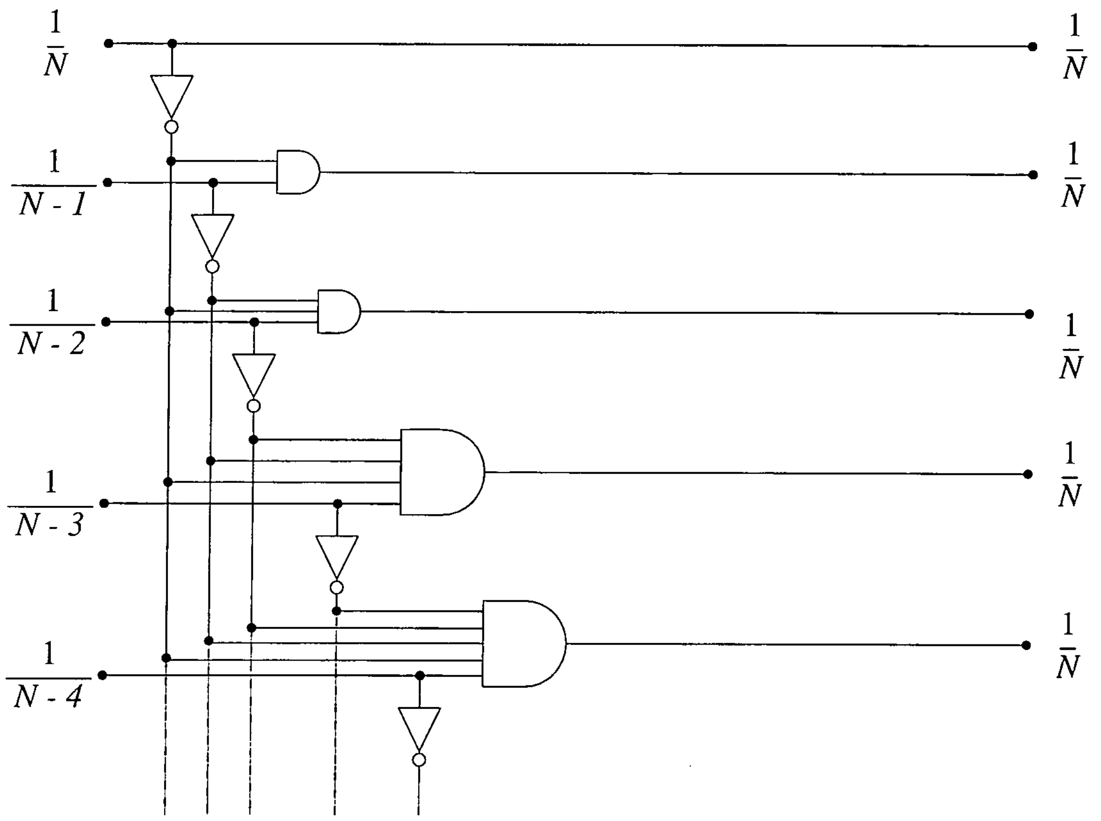


Figure 4.11: Initial circuit for the generation of N pulse streams of value $\frac{1}{N}$. *Note the large loading placed upon inverters at the top of the cascade and the large number of inputs for the AND gate at the bottom of the cascade.*

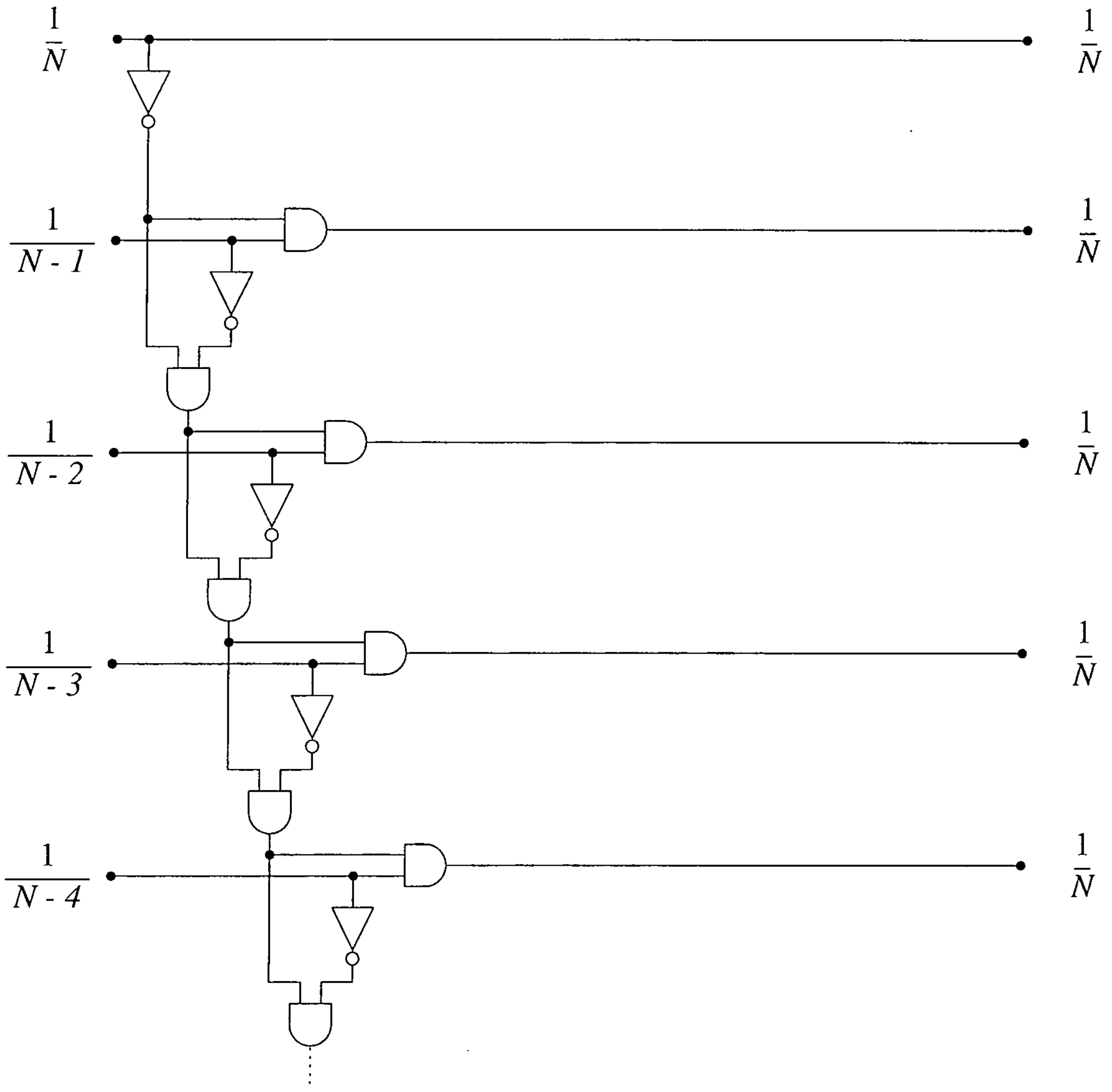


Figure 4.12: Improved circuit for the generation of N pulse streams of value $\frac{1}{N}$. *Note the modest and consistent fan-out and fan-in for all stages of the circuit.*

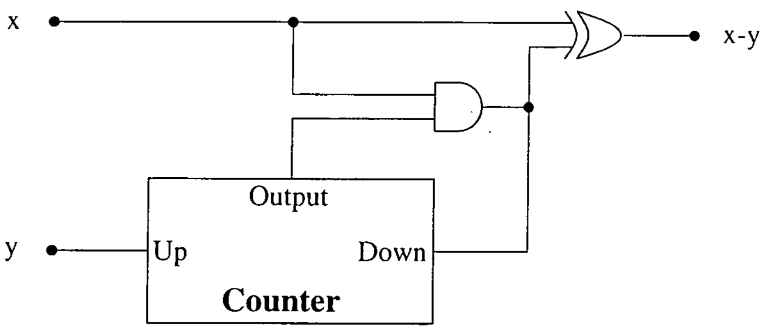


Figure 4.13: SLU subtraction by pulse removal. *In this circuit y is subtracted from x by counting the pulses on y and by means of the* **AND** *and* **XOR** *gates detecting and removing the pulse from x.*

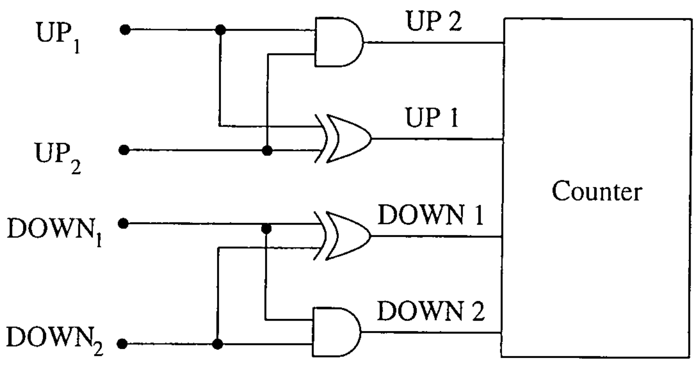


Figure 4.14: Two input summing integrator for DLB. *This circuit requires a counter which will increment and decrement by two. The circuit performs equally weighted integration of the two DLB inputs.*

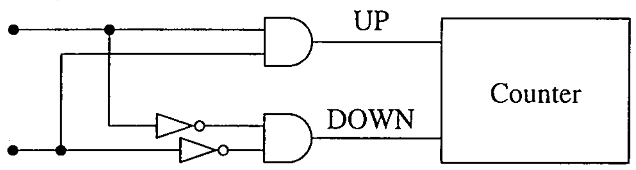


Figure 4.15: Two input summing integrator for SLB.

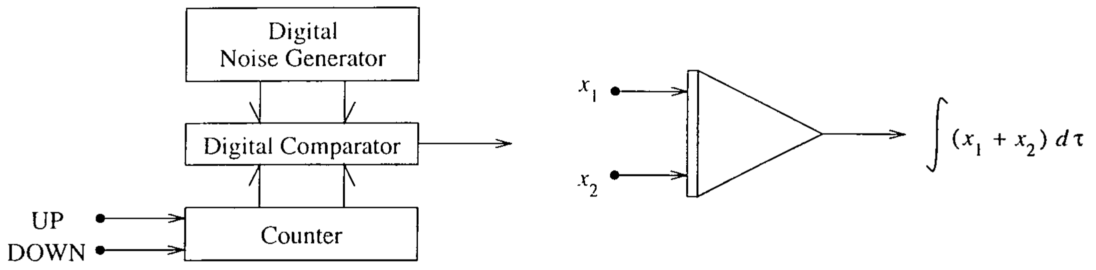


Figure 4.16: Generic two input summing integrator. *This circuit performs integration of the two input signals and re-encodes the resultant deterministic value into the stochastic pulse rate encoded domain.*

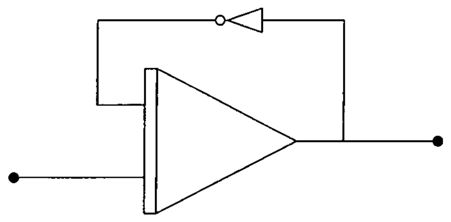


Figure 4.17: Schematic of an ADDIE. *The ADDIE is used as the basis for output interfaces.*

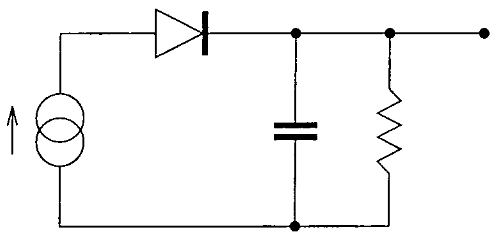


Figure 4.18: Schematic of a frequency modulation detector. *For a source of fixed frequency input pulses the output will be a steady state voltage dependant on the input frequency. This will occur when the rate of charging of the capacitor by the pulse stream is equal to the rate of discharge through the resistor.*

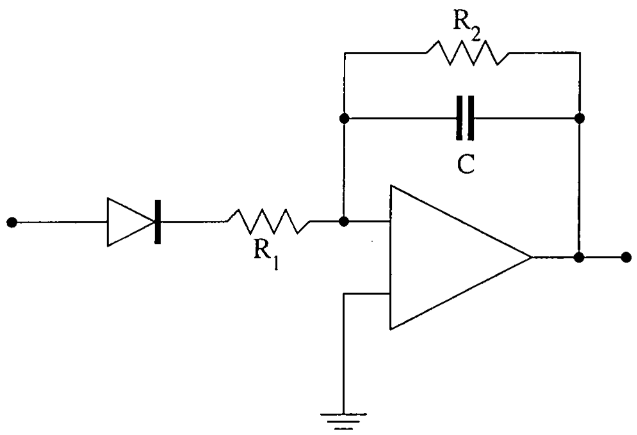


Figure 4.19: Schematic of an analogue frequency modulation detector. *The output voltage for an input pulse stream is dependant upon the ratios of the resistors.*

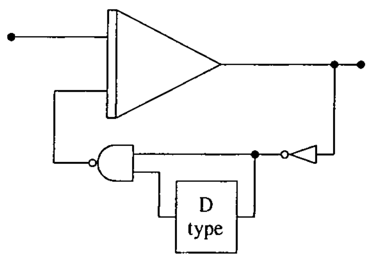


Figure 4.20: ADDIE circuit to obtain the square-root of a pulse stream. *The square of the inverse of the output is fed back in this configuration.*

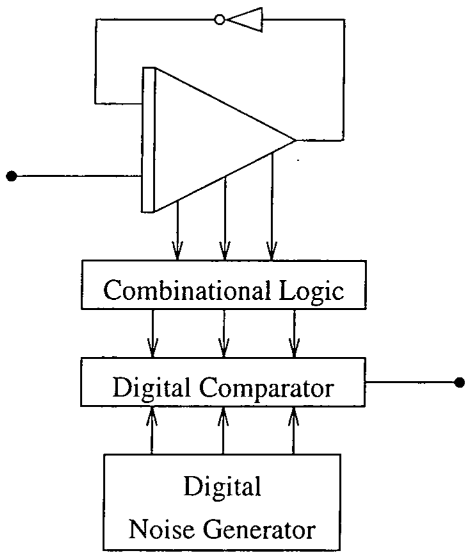


Figure 4.21: Generic ADDIE circuit to obtain arbitrary function transformations.

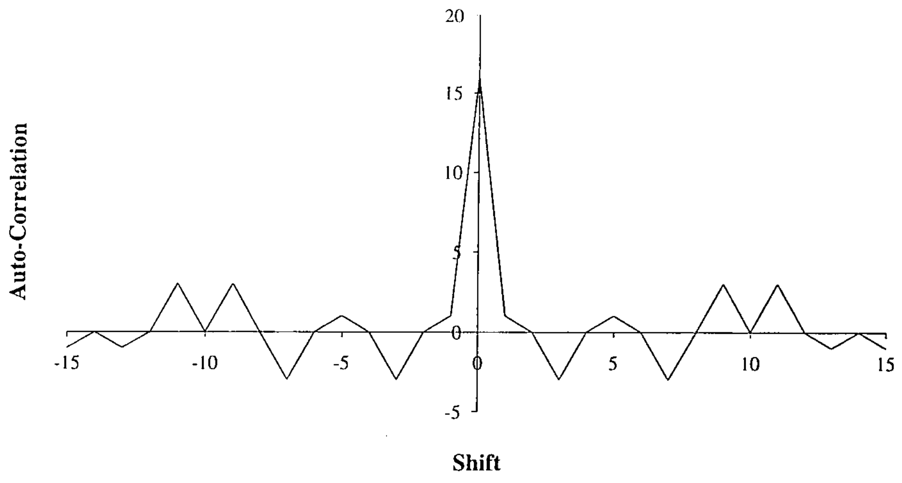


Figure 4.22: 16-bit e-sequence autocorrelation function. *All even shifts, except zero, produce a zero result.*

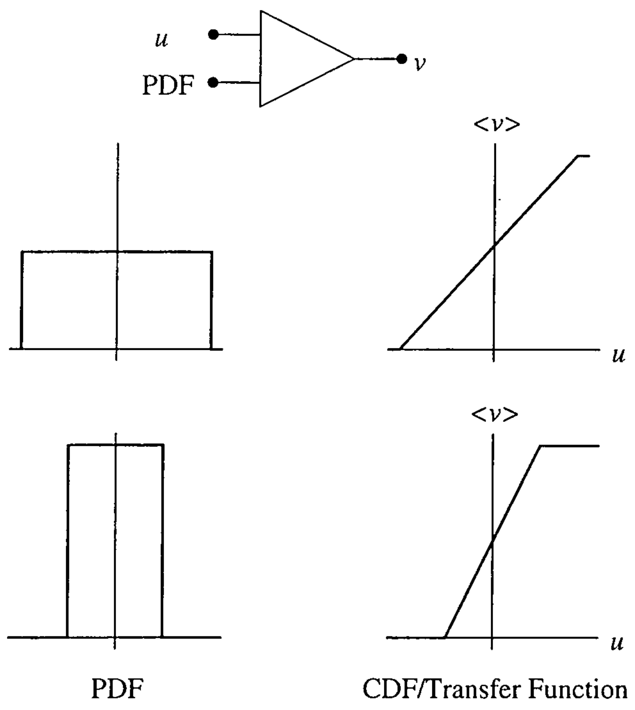


Figure 4.23: PDFs with associated CDFs for a URN. *Adjusting the probability density function (PDF) distribution varies the cumulative distribution function (CDF) distribution given a uniform random number (URN).*

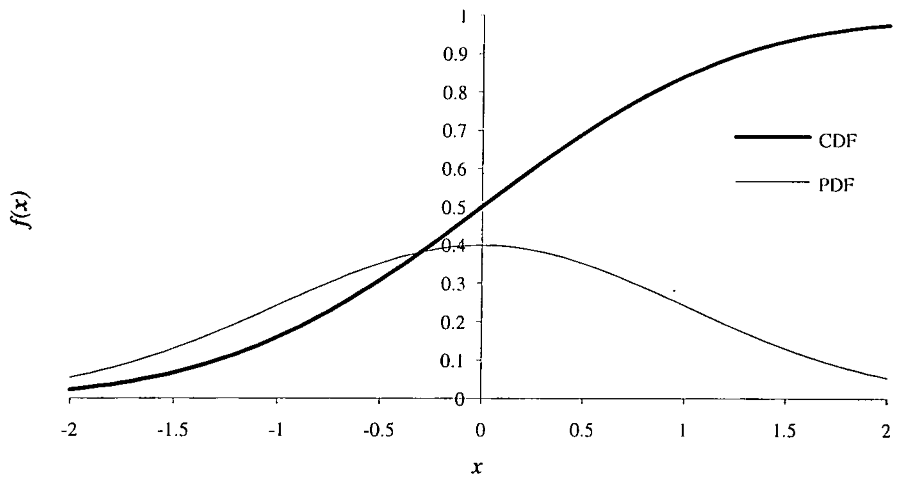


Figure 4.24: PDF with associated CDF for a Gaussian random number. *The mean and variance for the PDF are 0 and 1 respectively.*

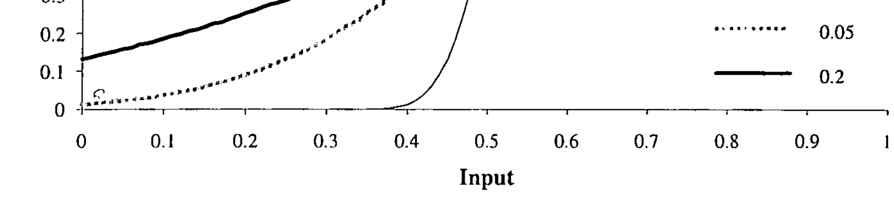


Figure 4.25: Sigmoids for adjusted variance values. *Decreasing the variance of the generating PDF increases the sharpness of gradient of the CDF.*

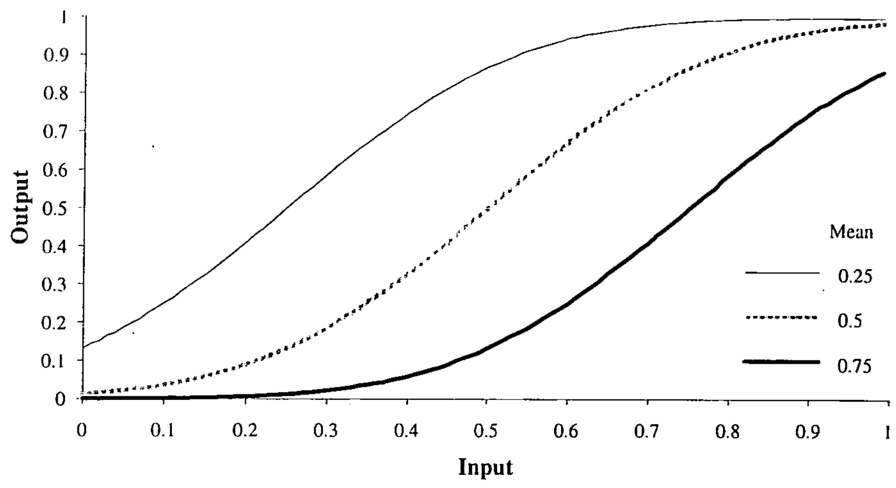


Figure 4.26: Sigmoids, resultant CDFs, for adjusted mean values of the generating PDF. *Increasing the PDF mean moves the sigmoid to the right, while decreasing the PDF mean shifts the sigmoid to the left.*

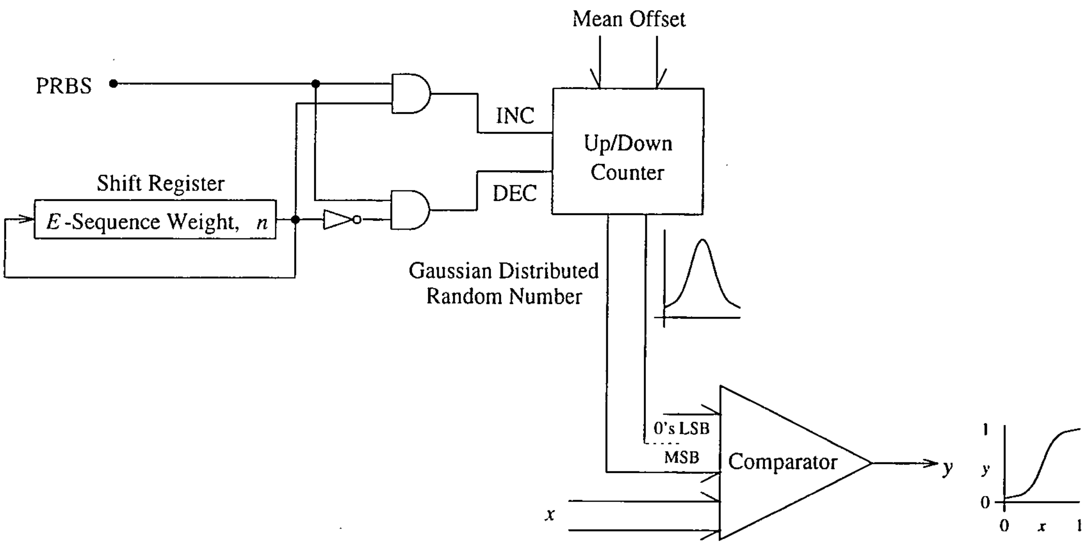


Figure 4.27: Sigmoidal transform generating circuit. *A Gaussian Random Number (GRN) is generated using an e-sequence. The GRN is compared to the input value, x, to produce the probability of a pulse output according to a sigmoidal transform.*

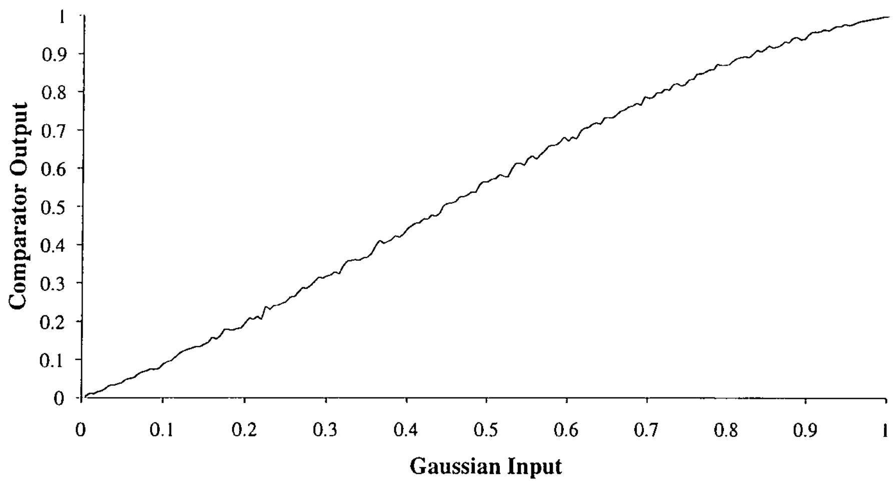


Figure 4.28: Sigmoid produced by encoding circuit simulation. *The sigmoid produced is only slight but it does exist. A more pronounced sigmoid could be produced by a GRN with a greater dynamic range, a larger e-sequence.*

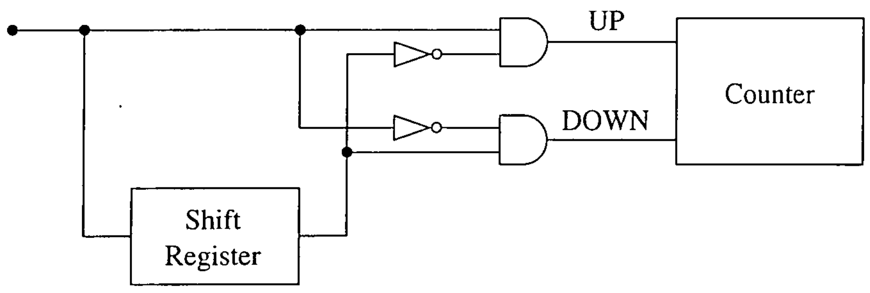


Figure 4.29: Moving average circuit implementation. *The shift register is used to hold the N previous sequence samples. Compare this circuit to that of Figure 4.15.*

Chapter 5

Multiple Random Number Generation

5.1 Introduction

The previous chapter, §4, has discussed a pulse rate computation technique using stochastic pulse rate encoded signals. This technique relies heavily upon a noise or random number source for encoding deterministic information into the stochastic pulse rate domain. A simple efficient technique for the generation of noise or random numbers for the purpose of encoding is needed. Since stochastic pulse rate computation operates digitally it would be preferable if the random number generator also operated by the use of digital circuits, it could then be fabricated in the same format as the rest of the processing structure. Many signals will need to encoded therefore the generation of multiple numbers will be investigated.

In this chapter a short review of techniques and implementation of random number generators is made together with possible tests which may be applied to the resulting sequence to assess their quality. Particular attention is paid to a class of generators known as Pseudo Random Binary Sequence (PRBS) generators from which it is possible to obtain more than one random number at a time. A technique is discussed for forming multiple sequences from a single PRBS. The technique leaves open ended the final stage of the selection of the appropriate circuits for sequence formation. The optimisation and search techniques of simulated annealing and genetic algorithms were applied to the selection process. It will be demonstrated that, in general, provided either algorithm is suitably configured it can be used for determination of the necessary circuits.

5.2 Generation of Random Numbers

A random number is a number, possibly within a specified range, which has no prearranged order and its value can not be determined in advance. A random number may be described probabilistically. For uniformly distributed random numbers each number has an exactly equal chance of being selected, but other distributions may be produced, eg. Gaussian, Poisson. Random numbers are required for many applications for modelling and simulation of processes, selection of input patterns to a neural network during a training phase, even the selection of a Premium Bond winner.

Random numbers can be generated either by using hardware or software algorithms. Hardware random number generators are frequently specialised pieces of equipment not usually suitable for integration into a general process. They are based upon naturally occurring random physical processes and produce excellent results. Software random number generators are algorithms that require direct calculation within a computer. They can be manipulated easily and are often implemented as functions or subroutines. Many computer languages have a random number function included in a standard library if not the main language eg. functions `rand()` and `drand48()` in C. A user should be aware that the quality of these functions can often leave much to be desired. Software random number generators do possess the advantageous property of repeatability by resetting the seed of the generator.

A description of hardware and software random number generators together with the tests which may be applied to them is given in Appendix A.

5.3 Pseudo Random Binary Sequence Generators

PRBSs are formed using digital circuits constructed from Linear Feedback Shift Registers, LFSRs. The feedback applied to the shift register determines the type of sequence formed. The type that is of interest in this case is that which performs modulo two arithmetic, **XOR** gates being used to achieve this.

5.3.1 Basic PRBS Generator Considerations

A shift register is a cascade connection of binary memory elements controlled in such a way that the contents may be transferred, shifted, along the register by applying an external clock pulse. Usually the direction of shift is fixed, although bi-directional shift registers exist. In practice a shift register is formed from an array of flip-flops in series. The output **Q** of each stage drives the input **D** of the following stage. The clock inputs of each stage are driven simultaneously, Figure 5.1

The size of a shift register with N stages is said to be of degree n or of order n. When clocked the contents of stage q_i moves into stage q_{i+1}. If no connection is made to the nth stage output back to the input, its contents are lost from the register. The value the first stage adopts depends upon the value its input is set to. The register holds n digits into the past and can be said to have a memory span of n.

If feedback from later stages is introduced to supply the input value to the first stage the future values of the shift register depend upon the present state of the register and the format of the feedback, Figure 5.2. For example, if the output of the last stage is fed directly back into the first stage an n-bit ring counter can be formed, or if the output of the last stage is fed back inverted an n-bit twisted ring counter can be formed. It is the configuration of the feedback for the shift register that is of interest to the generation of random numbers. The feedback network, $f(x_1, x_2, \cdots, x_n)$, may be any combination of binary logic function.

Tausworthe, [85], developed a random number generator based upon the above principle of linear feedback. Modulo 2 arithmetic is applied to the feedback, ie. **XOR** gates are used to form the feedback network. Appropriately selected feedback on the shift register will enable an output bit sequence of length $2^n - 1$, maximal length known as an m-sequence. The feedback configuration for a Tausworthe generator is described by its characteristic equation

$$D^n \oplus D^{n-s} \oplus 1 = 0$$

where D is the delay operator, n is the length of the generator and s is an output from another stage in the shift register. The PRBS configuration may also be described in terms of the feedback stages, x^p, used to generate the next bit in the sequence to be moved into the register,

$$x^{n-1} \oplus x^{n-s-1} = x^{-1}$$

The characteristic equation is a primitive polynomial, ie. it is an irreducible polynomial. Other **XOR** feedback combinations can be used but the sequence will not necessarily be maximal length. Tables of irreducible polynomials have been published to reduce the need to calculate them, [86]. During production of the bit stream the shift register will cycle once through all its possible states, except the all zeros state, before repeating. The all zero state is self replicating. The sequence of bits output is a Bernoulli sequence of probability 0.5. Since the sequence length is odd the number of 1's and 0's will vary by only one, the number of consecutive logic levels of a particular state is directly related to the length of the run, ie. half the runs will be of length one, a quarter of length two, an eighth of length three, etc.

The realisation of a PRBS can be achieved efficiently in software by a few lines of code, but for the fast generation of values a hardware method is preferable. Several architectures have been used from a simple single shift register to more elaborate schemes using multiple shift registers, [87, 88, 89], the latter allowing increased speed in the formation of random numbers when many steps are required to advance the generator beyond correlation.

5.3.2 Delayed PRBSs

Having produced a single pseudo random bit stream, how can multiple instances be generated which can be considered independent? If the sequence is sufficiently long then the autocorrelation between delayed versions is small except where the two sequences are synchronous. For an n-stage binary shift register generator a maximal length sequence the normalised autocorrelation function for a period L bits is given by

$$A(k) = \frac{1}{L}\sum_{L} y_i y_{i+k}$$

k denotes discrete time delay, and the sequence is expressed as $+1$ and -1 rather than 1 and 0. The transformation from x_i to y_i is given by

$$y_i = (-1)^{x_i} = 1 - 2x_i$$

$$1 \rightarrow -1, 0 \rightarrow 1$$

The autocorrelation function has the appearance of Figure 5.3. It can be seen that for all except synchronous sequences the correlation is negligible and they can be considered as independent sequences. It is feasible to have g generators each of the same feedback configuration but with a different seed state producing g sequences. This method is inefficient in its realisation requiring the formation of many generators.

Viewing the configuration of a single PRBS generator it can be seen that adjacent cells of the register will cycle through the same sequence as that produced by the output but delayed by the appropriate number of bits. A single PRBS could be produced with multiple cells after the base n cells to store the delayed sequence in, Figure 5.4. For sequences of long length and large delays the overall shift register length will become prohibitive for practical formulation.

Tsao, [1], demonstrated how, using modulo two arithmetic and the shift-and-add property of m-sequences, specified delayed versions of a sequence can be realised. Figure 5.5 illustrates the initial steps needed. It can be seen that the number of **XOR** gates needed depends upon the delay and number of serial additions required. The overall speed of operation of the generator will be hampered by the propagation delay through the **XOR** gate tree. A problem is to determine the necessary tap combinations for a given delay. This problem has been resolved in several ways.

Tsao resolves the problem of determining the required tap points by manipulation of the characteristic equation,

$$D^{n-1} \oplus D^p \oplus \cdots \oplus D^r \oplus D^s = D^{-1}$$

$$\Rightarrow D^n \oplus D^{p+1} \oplus \cdots \oplus D^{r+1} \oplus D^{s+1} \oplus D^0 = 0$$

This is best illustrated using the three Tsao examples for a four-stage PRBS. These three cases will be reiterated for other techniques which have been developed.

For a four-stage PRBS the characteristic equation is;

$$D^3 \oplus D^2 = 1 \text{ or } D^4 \oplus D^3 \oplus D^0 = 0$$

The delay combinations for D^5, D^6 and D^{13} are to be deduced.

1. D^5

   Rearranging the characteristic equation,

$$D^4 = D^3 \oplus D^0$$

$$\begin{aligned} D^5 = DD^4 &= D(D^3 \oplus D^0) \\ &= D^3 \oplus D \oplus D^0 \end{aligned}$$

2. D^6

$$\begin{aligned} D^6 = D^2D^4 &= D^2(D^3 \oplus D^0) \\ &= D^5 D^2 \\ &= D^3 \oplus D^2 \oplus D \oplus D^0 \end{aligned}$$

3. Finally D^{13}

$$D^{13} = D^{-2}$$

   Extract D^2 from characteristic equation,

$$\begin{aligned} \Rightarrow D^2(D^2 \oplus D \oplus D^{-2}) &= 0 \\ D^2 &\neq 0 \\ D^2 \oplus D \oplus D^{-2} &= 0 \\ D^{-2} &= D^2 \oplus D \end{aligned}$$

The mathematical manipulations necessary for each individual delay are not always obvious. As the characteristic equation and delays desired become longer the modulo two algebra becomes more demanding.

Davies, [90], observed that if the required delay, D^j, is divided into the characteristic equation, $f(D)$, then

$$\frac{D^j}{f(D)} = q(D) \oplus r(D)$$

where $q(D)$ is the quotient and $r(d)$ is the remainder, ie.

$$D^j = f(D)q(D) \oplus r(D)$$

For the m-sequence

$$f(D) = 0 \Rightarrow f(D)q(D) = 0$$

thus

$$D^j = r(D)$$

The coefficients of the remainder, $r(D)$, are the desired tap off points from the shift register. Practical considerations for the calculation of $r(D)$ are considered by Davies, [91] and Van Luyn, [92]. The division technique will now be used to calculate the connections for the previous three cases.

1. D^5

$$\begin{array}{ccccccccccccc}
4 & 3 & 2 & 1 & 0 & & 5 & 4 & 3 & 2 & 1 & 0 \\
 & & & & & & & & & & & \\
1 & 1 & 0 & 0 & 1 &) & \overline{1} & \overline{0} & \overline{0} & \overline{0} & \overline{0} & \overline{0} \\
 & & & & & & \underline{1} & \underline{1} & \underline{0} & \underline{0} & \underline{1} & \\
 & & & & & & & 1 & 0 & 0 & 1 & \\
 & & & & & & & \underline{1} & \underline{1} & \underline{0} & \underline{0} & \underline{1} \\
 & & & & & & & & 1 & 0 & 1 & 1
\end{array}$$

$$\begin{array}{rlllll}
D^4 \oplus D^3 \oplus D^0 \;) & \overline{D^5 \quad\quad\quad\quad\quad\quad\quad\quad\quad} & & & & \\
 & \underline{D^5 \oplus D^4 \oplus} & & \underline{D^1} & & \\
 & & D^4 \oplus & D^1 & & \\
 & & \underline{D^4 \oplus D^3 \oplus} & & \underline{D^0} & \\
 & & & D^3 \oplus D^1 \oplus D^0 & &
\end{array}$$

2. D^6

```
4 3 2 1 0   6 5 4 3 2 1 0
            _____________
1 1 0 0 1 ) 1 0 0 0 0 0 0
            1 1 0 0 1
            ---------
              1 0 0 1
              1 1 0 0 1
              ---------
                1 0 1 1
                1 1 0 0 1
                ---------
                  1 1 1 1
```

$$\begin{array}{rccccccc}
\cline{2-8}
D^4 \oplus D^3 \oplus D^0\,) & D^6 & & & & & & \\
 & D^6 \oplus & D^5 \oplus & & & D^2 & & \\ \cline{2-6}
 & & D^5 \oplus & & & D^2 & & \\
 & & D^5 \oplus & D^4 \oplus & & & D^1 & \\ \cline{3-7}
 & & & D^4 \oplus & & D^2 \oplus & D^1 \oplus & D^0 \\
 & & & D^4 \oplus & D^3 \oplus & & & D^0 \\ \cline{4-8}
 & & & & D^3 \oplus & D^2 \oplus & D^1 \oplus & D^0
\end{array}$$

3. D^{13}

```
4 3 2 1 0   13 12 11 10 9  7  8  6  5  4  3  2  1  0
            ________________________________________
1 1 0 0 1 ) 1  0  0  0  0  0  0  0  0  0  0  0  0  0
            1  1  0  0  1
            -------------
               1  0  0  1
               1  1  0  0  1
               -------------
                  1  0  1  1
                  1  1  0  0  1
                  -------------
                     1  1  1  1
                     1  1  0  0  1
                     -------------
                           1  1  1
                           1  1  0  0  1
                           -------------
                                 1  0  1
                                 1  1  0  0  1
                                 -------------
                                    1  1  0  1
                                    1  1  0  0  1
                                    -------------
                                             1  1
```

$$
\begin{array}{rl}
D^4 \oplus D^3 \oplus D^0 \;) & \overline{D^{13} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
 & \underline{D^{13} \oplus D^{12} \oplus \qquad\quad D^9} \\
 & \quad D^{12} \oplus \qquad\quad D^9 \\
 & \quad \underline{D^{12} \oplus D^{11} \oplus \qquad D^8} \\
 & \qquad D^{11} \oplus \qquad D^9 \oplus D^8 \\
 & \qquad \underline{D^{11} \oplus D^{10} \oplus \qquad\quad D^7} \\
 & \qquad\quad D^{10} \oplus D^9 \oplus D^8 \oplus D^7 \\
 & \qquad\quad \underline{D^{10} \oplus D^9 \oplus \qquad\quad D^6} \\
 & \qquad\qquad\qquad D^8 \oplus D^7 \oplus D^6 \\
 & \qquad\qquad\qquad \underline{D^8 \oplus D^7 \oplus \qquad\quad D^4} \\
 & \qquad\qquad\qquad\qquad D^6 \oplus \qquad D^4 \\
 & \qquad\qquad\qquad\qquad \underline{D^6 \oplus D^5 \oplus \qquad D^2} \\
 & \qquad\qquad\qquad\qquad\quad D^5 \oplus D^4 \oplus D^2 \\
 & \qquad\qquad\qquad\qquad\quad \underline{D^5 \oplus D^4 \oplus \qquad D^1} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\quad D^2 \oplus D^1
\end{array}
$$

Gardiner, [93], provides a third general purpose method for determining feedback combination delays. The basic principle is to increment all the delays in the characteristic equation by one and when a delay is produced that is outside the bounds of those which can be directly obtained from the generator to reduce the equation to terms which can. Illustration by example is probably the best method to understand this technique, therefore repeating for the last time the example generator for delays D^5, D^6 and D^{13} we have the characteristic equation in the form

$$D^3 \oplus D^2 = 1$$

$$D^4 = D^3 \oplus D^0 \tag{5.1}$$

1. D^5 Increment delays

$$D^5 = D^4 \oplus D^1$$

D^4 cannot be obtained from the PRBS directly, but substituting from eq.(5.1) produces:

$$D^5 = D^3 \oplus D^1 \oplus D^0$$

2. D^6 Again increment the previous equation

$$D^6 = D^4 \oplus D^2 \oplus D^0$$

Again using eq.(5.1) this can be reduced to the minimal configuration

$$D^6 = D^3 \oplus D^2 \oplus D^1 \oplus D^0$$

3. D^{13} Increment delay by three from that of D^6

$$\begin{aligned} D^9 &= D^6 \oplus D^5 \oplus D^4 \oplus D^3 \\ D^9 &= (D^3 \oplus D^2 \oplus D^1 \oplus D^0) \oplus (D^3 \oplus D^1 \oplus D^0) \oplus (D^3 \oplus D^0) \oplus D^3 \end{aligned}$$

Note that $D^i \oplus D^i = 0$

$$D^9 = D^2 \oplus D^0$$

Increment delay by four

$$\begin{aligned} D^{13} &= D^6 \oplus D^4 \\ &= (D^3 \oplus D^2 \oplus D^1 \oplus D^0) \oplus (D^3 \oplus D^0) \\ &= D^2 \oplus D^1 \end{aligned}$$

In the last example, D^{13}, rather than increment by a single delay and reduce the subsequent equation, an increment of multiple delays is used before reduction of the resulting equation.

It can be seen from the above three techniques that any one may be suitable to find a tap combination to produce a single delay of the fundamental sequence. However, it is not possible prior to the calculation to determine how many tap off points will be required for a delay and where they will lie. In addition, if several delayed sequences are required, many taps from a single shift register cell may be required causing uneven loading upon the shift register. Considering these points the following section considers a possible solution to these problems of forming multiple PRBS sequences from a single generator.

5.3.3 Multiple PRBS

The methods described above for obtaining the tap pattern required for a single delay are in general adequate for most needs. If multiple pulse sequences are required from a single generator these techniques are no longer practical since other considerations besides absolute delay must be considered. Firstly, the number of taps which must be **XOR**ed to form a delay must be a reasonable size. If this fan-in is too large it will result in complications when attempting to connect up a circuit. The algorithms of section §5.3.2 do not provide any knowledge of the number of taps required to form a delay prior to their calculation. Secondly, if the number of delays which require a tap from a given element of the shift register is too large the loading will adversely affect the shift register performance. Alspector *et al*, [94], highlights these problems and offers a possible solution.

Alspector's solution for resolving the dilemma, the basic principle of which is the reverse of the methods outlined in §5.3.2, has been implemented in software. The technique will now be outlined. Groups or buckets of tap combinations are first formed that satisfy the following three constraints.

1. The number of taps required to produce a delay, F, is bounded eg.

   $$2 \leq F \leq 5$$

   F is used since it represents the fan-in to the **XOR** gate necessary to produce the delay.

2. The delay, d, created by a given tap combination is to be within a given bound of the optimal value, D.

   $$d = D \pm \delta$$

   δ is the delay tolerance.

3. The loading, L, placed upon elements of the shift register shall be evenly distributed and as low as possible.

After generating all tap combinations which satisfy condition (1) and placing them in buckets where their associated delays satisfy (2), it is then necessary to select from each bucket a tap combination which minimises a cost function based upon all three constraints. Note, not all tap combinations which satisfy (1) will have a suitable delay. The number of possible combinations of selection from each bucket will be large. Hypothetically, for 31 equally spaced delayed sequences 31 buckets would be required, if each contained just two tap patterns the number of configurations to evaluate is $2^{31} = 2147183647$. In practice there will be more tap patterns per bucket and an even larger search space. An exhaustive search of all these possible solutions is prohibitive in the amount of computation time required. In Alspector *et al* [94] paper it was suggested that the search process may be conducted by random or deterministic techniques and possibly simulated annealing. Thus, in section §5.4 and §5.5 a discussion of the implementation of two random search algorithms, Simulated Annealing and Genetic Algorithms, is made. These two algorithms were experimented with to find an optimal form for the taps from all the possible combinations. An example of Alspector's system may best illustrate the whole procedure.

Given a PRBS generator of 10-bits from which we require five sequences the nominal spacing between delays is:

$$\frac{m\text{-sequence length}}{\text{Number of sequences}} = \frac{2^n - 1}{5} = 204.6$$

Therefore delays of 0, 205, 410, 615 and 820 are required. A range of taps that are suitable is specified. The range two to four will be used in this case. It will be observed that a delay difference of ± 1 can be achieved by moving a tap up/down the shift register, Figure 5.6. Similarly by moving complete tap patterns up and down the shift register the overall delay can be adjusted, Figure 5.7. A set of essential tap patterns can be defined where a single tap is always the least/most significant bit. Near delays are determined by shifting the tap patterns by p and adjusting the delay by p. For two taps per pattern the essential taps are illustrated in Figure 5.8. By extrapolation the principle can be expanded to any other number of taps.

The number of tap patterns for a given number of taps is

$$\begin{pmatrix} N \\ K \end{pmatrix} \triangleq \frac{N!}{N!(N-K)!} \tag{5.2}$$

whereas the number of essential tap patterns is

$$\begin{pmatrix} N-1 \\ K-1 \end{pmatrix} \triangleq \frac{(N-1)!}{(K-1)!(N-K)!} \tag{5.3}$$

Here N is the length of the PRBS generator and K is the number of taps to be used. For the example of the 10-bit PRBS with the range of taps from two to four the total number of tap patterns is

$$\sum_{K=2}^{4} \begin{pmatrix} 10 \\ K \end{pmatrix} = 375$$

but the number of essential tap patterns is

$$\sum_{K=2}^{4} \begin{pmatrix} 9 \\ K-1 \end{pmatrix} = 129$$

A table of essential tap patterns will exist for two, three and four taps per pattern. The correct delay must now be associated with these patterns. Three methods are proffered by Alspector for the solution of tap patterns and delays, the Simple Shifting Method, the Giant Step/Baby Step Method and the Discrete Logarithm Method. The techniques increase the speed of association of a delay with a pattern but also increases the complexity of the implementation. The Simple Shifting Method was adopted for ease of implementation and will be detailed, refer to Alspector's paper for details of the other two methods.

For a given tap combination from the PRBS determine the output that produces from n clocks of the PRBS. n is the length of the PRBS. This n bit output vector, g, is stored. The PRBS is reset to its initial base value and clocked. An n bit rolling output vector from the PRBS is maintained. This rolling output vector is compared with the vector

g. When these two vectors are equivalent the number of clock cycles required is the shift associated with the tap combination.

Having calculated the delay for each of the 129 essential tap combinations the table of delay/tap pairs can be expanded to cover all 375 tap combinations. A tolerance band is placed round each of the nominal delays to create a bucket into which delay/tap pairs are placed. For a tolerance of ± 50 the delay buckets are illustrated in Table 5.1.

Lower Delay Limit	Nominal Delay	Upper Delay Limit
974	0	50
155	205	255
360	410	460
365	615	665
770	820	870

Table 5.1: Delay Buckets for Five Delays From a 10-bit PRBS with a Tolerance of ± 50

A search is made of selections from each bucket of delay/tap combinations to find the most suitable.

Thus it can be seen that multiple PRBS m-sequences can be formed from a single PRBS generator. Actually it is the same m-sequence viewed at different instances. Providing the length of each m-sequence used at any time is not too long, ie. a sequence does not overlap with another, the degree of correlation will be low. These sequences from the PRBS may be used as separate noise sources.

5.3.4 PRBS to Random Number Conversion

To be able to utilise a PRBS sequence as a random number it must be correctly converted from a series of bits. The basic technique is to form a sequence of bits output by the PRBS generator into a digital word and to treat this word as a random value. To form subsequent random values the generator is advanced so that new random bits are advanced into the register holding the digital word. It is necessary to advance the generator by more than the size of the digital word otherwise a correlation will exist between random values, Figure 5.9.

5.4 Simulated Annealing

Simulated Annealing, SA, is an optimisation process with its roots based on the processes of annealing within condensed matter physics. The analogy made is with thermodynamic processes. For example, at the start of the annealing process the matter will be at a high temperature and in a fluid phase. The fluid is allowed to cool slowly so that the molecules are able to align themselves as thermal mobility is lost. Cooling further will enable the formation of crystals and solids as the state of minimum energy for the system is found. As the temperature tends towards zero so the energy of the system tends to a minimum. More specifically, [95], at a given temperature, T, when thermal equilibrium has been reached the material state can be characterised by the probability of it being in a state with energy, E, given by the Boltzmann Distribution.

$$Pr\{\mathbf{E} = E\} = \frac{1}{Z(t)} e^{-\frac{E}{k_B T}} \tag{5.4}$$

$Z(t)$ is known as the partition function and acts as a normalisation function dependent upon the temperature. The term $e^{-\frac{E}{k_B T}}$ is the Boltzmann Factor, where k_B is the Boltzmann constant. Slowly decreasing the temperature concentrates the Boltzmann distribution into the state with the least energy. As the temperature approaches zero only the minimum energy state has a non-zero probability of occurrence.

Metropolis *et al*, [96], modelled the annealing process in matter. Using a Monte Carlo method to select the sequence of states for the matter, a state being characterised by the position of the particles of matter, the energy of the configuration was calculated. A new state was generated by a random perturbation of the existing state. The amount of perturbation depends on the temperature of the system, a higher temperature causing a greater disturbance, the difference in energies, ΔE, between the existing state and the new state being used as a basis for determining if the new state should be maintained. If $\Delta E < 0$, ie. a decrease of energy in the system, the new state is kept and used as the base for restarting the cycle. If $\Delta E \geq 0$ the acceptance of the new state is probabilistic. The probability of acceptance is $e^{-\frac{\Delta E}{k_B T}}$

$$p(accept) = \begin{cases} 1 & \text{if } \Delta E < 0 \\ e^{-\frac{\Delta E}{k_B T}} & \text{if } \Delta E \geq 0 \end{cases} \tag{5.5}$$

therefore it is possible for a new present state to be reached with a higher energy requirement.

This acceptance rule is the Metropolis criterion. Repeating the perturbation process many times results in a distribution approaching that of a Boltzmann distribution. The entire process is known as the Metropolis Algorithm.

Transferring the idea of annealing to general optimisation problems requires the association of temperature, energy and state within the new domain. This was first achieved by Kirkpatrick *et al*, [97], in their application to the physical design of computers eg. integrated circuit placement and wiring routes. Subsequently the technique has been widely applied. The state in the new domain is the organisation, configuration or set of values taken to represent that state. To this configuration is assigned a cost function, C, which represents the amount of energy within the system, the aim is to minimise the value of the cost function. Temperature is represented by a control parameter, c, which initially has a high value. For a randomly selected combination of system parameters, configuration i, the cost function is evaluated, $C(i)$. A random selection of new elements in the neighbourhood of i is made, configuration j, for which the cost is also evaluated, $C(j)$. Whether or not this new configuration is accepted as the basis for further improvements depends on the Metropolis criterion applied to the difference in costs, ΔC_{ij}.

$$\Delta C_{ij} = C(j) - C(i) \tag{5.6}$$

The probability that configuration j is used as the next base configuration is,

$$p(accept) = \begin{cases} 1 & \text{if } \Delta C_{ij} < 0 \\ e^{-\frac{\Delta C_{ij}}{c}} & \text{if } \Delta C_{ij} \geq 0 \end{cases} \tag{5.7}$$

If $\Delta_{ij} \geq 0$ it is possible for a new configuration to be reached with a higher cost function value associated with it.

The value of c is reduced in steps, the system being allowed to reach an equilibrium at each value of the control parameter. The algorithm is stopped when the control parameter reaches a predetermined small value. Simulated annealing is thus a series of applications of the Metropolis algorithm for decreasing values of c. As an alternative the control parameter is reduced continuously with time rather than in steps. The above two formats divide simulated annealing into two categories, [95], the former an homogeneous algorithm which can be described by a series of homogeneous Markov chains, and the latter an inhomogeneous algorithm described by one inhomogeneous Markov chain.

Applying the simulated annealing technique to optimising the tap combinations selected for the PRBS generator a cost function must be defined. Relevant parameters to be considered in this function are the number of taps required to form a delay, F, the loading the delay configurations places upon the shift register elements, L, and finally the distance, d, of a delay from its nominal delay.

$$C = f(F) + g(L) + h(d) \tag{5.8}$$

For the generic cost function, Equation 5.8, a low cost must be produced for favourable configurations and a high cost for unfavourable ones. For $f(F)$ the less taps required to form a delay the simpler the **XOR** gate required, while the function for the loading placed upon individual shift register elements, $g(L)$, the more evenly distributed the taps are across all elements of the shift register the better. Non-linear penalties were applied to these factors such that a small increase in the number of taps required for a delay, F, or the overall loading placed on a shift register element, L, becomes increasingly expensive. For the cost factor attached to the distance of the actual delay selected from the desired nominal delay, $h(d)$, it was found that very large differences in the delay were necessary which outweighed the combined cost of $f(F)$ and $g(L)$, therefore the difference in delay, d, was scaled down to a similar order of magnitude. The delay difference is still accounted for but is not the predominant concern. The resulting specific cost function is

$$C = \sum_{x=1}^{X} (F_x)^2 + \sum_{y=1}^{Y} (L_y)^3 + \sum_{x=1}^{X} \frac{d}{1000} \tag{5.9}$$

where X is the number of m-sequences required from the shift register and Y is the number of elements which make up the shift register.

The simulated annealing technique was applied in two ways which varied in the amount of perturbation the system received, the cooling schedule and the Metropolis criterion.

Scheme 1. From an initial random state with a known cost a new state is formed by selecting at random a delay for each m-sequence in turn. After each m-sequence has been adjusted the cost of the configuration is calculated. The Metropolis criterion is applied where $p(accept)$ is tested against a control parameter 'warmth'. 'warmth' is decreased at regular intervals but has no bearing on the amount the system is perturbed. Once all m-sequences have been subjected to adjustment the first one is revisited.

This variant of simulated annealing ensures that a new state is a close neighbour to the existing state since between two consecutive states 30 of the 31 m-sequences are the same.

Scheme 2. This second formulation of simulated annealing causes a greater disturbance of the configuration between one state and the next. Each element in the configuration is subjected to the possibility of change depending upon the value of 'warmth'. Initially, when 'warmth' has a high value many new m-sequences are selected for the next state, but as the system cools and 'warmth' is not as great less m-sequences alter between one state and the next.

The form of Metropolis criterion used for accepting or rejecting a state is dependent both on the change in cost and the value of 'warmth'. This method is more akin to Metropolis's, [96], and Kirkpatrick's, [97], implementation than the previous scheme.

Two sets of data were available to evaluate the performance of the above two schemes. The sets of data were 31 buckets of delays and associated tap patterns, where $\delta = 10000$ and $2 \leq F \leq 5$. The second set of data differed from the first in that in each bucket a delay existed which matched the optimal value, associated with the delay was a tap pattern of all zeros. This second set was to test the ability of simulated annealing to seek out a known global minimum for a given cost function, ie. each m-sequence would be for optimal delay and have no cost, likewise the all zero tap pattern would incur no cost either.

5.5 Genetic Algorithms

Genetic Algorithms, GA, are a type of optimisation technique which like simulated annealing have their roots in the natural world. Genetic algorithms take their lead from nature. In nature information about an organism is coded into the biological structure known as a chromosome. The information is stored in genes which are a constituent part of the chromosome. The value of the gene is known as an allele. For a species to evolve these chromosomes reproduce, crossover (chromosomes exchange section or genes) and mutate (a section of chromosome or an individual gene alters). During the life of the new organisms formed only the fittest will normally survive in a population of many varieties. Much of the early work in the field of genetic algorithms was conducted by Holland, [98].

For genetic algorithms a string is defined for the system which is an encoded description of the state of the system, a string being analogous to a chromosome. To determine the fitness of a string, ie. the set of conditions, for an environment a cost function is used similar to that used with the above simulated annealing technique. An individual string would be the same as a single state description in simulated annealing. Rather than just one string a population of strings is used each with an associated fitness value computed from the cost function. A new population is produced by selecting strings from the existing population with a probability proportional to the strings fitness. Strings with large fitness value have a higher probability of selection and are therefore more likely to survive the reproduction phase to the next generation. It is possible that a string will be replicated several times in the new population.

The next stage of the genetic algorithm is crossover. Two strings are selected at random from the child population. Within these two strings a common point is randomly selected and the two strings are exchanged at this point with a probability of crossover, P_c. Normally the value of P_c is quite high, eg. $P_c \geq 0.6$. This operation is the one point crossover and is illustrated in Figure 5.10 for binary encoded strings. Variations on this scheme can and have been used such as the n-point crossover and crossover between more than two strings at a time. The aim of crossover is to cause a blending of fit strings to produce fitter ones.

Finally in the genetic algorithm cycle each feature of each string is subjected to the possibility of mutation with probability P_m. A feature which is mutated has its value modified to another value within its parameter set. This modification is a random selection and may or may not include the features present value. The probability of mutation is usually quite low otherwise the entire algorithm would degenerate into a random search of available configurations. The purpose of mutation is to introduce diversification and new features into the population which may not be present in any of the parent strings.

The whole genetic algorithm cycle is restarted with this new population as the base for reproduction. Note, if crossover and mutation are pursued too aggressively salient feature groups may not be able to be sustained through generations.

The basic algorithm is simple, straightforward and has been found to be robust when applied to many combinational optimisation problems and searches of a result space. Overall genetic algorithms are distinguished from other optimisation techniques by the following properties,

1. direct manipulation of the coding.
2. search from a population of possible solutions, not from a single point.
3. search is conducted via sampling from a population, a blind search.
4. the search uses stochastic operators, not a deterministic process.

Similar to simulated annealing a cost function exists which is used to evaluate candidates produced by a pass through the algorithm. The basic algorithms operation proceeds in a very straight forward manner.

How then are genetic algorithms to be applied to the combinatorial optimisation problem of PRBS tap optimisation? Firstly a string must be designed to represent the tap patterns selected, secondly a cost function to evaluate the fitness of such a string must be defined. The string used is composed of a set of 31 numbers, each number representing one tap pattern from each of the tap buckets in sequence. Using this format the same cost function used to calculate the performance for simulated annealing can be used to drive the genetic algorithm, Equation 5.9. A look up table to correlate the tap pattern numbers in a bucket to an actual pattern is used.

The genetic algorithm is implemented as follows. From a set of parents a next generation of children is formed by selecting two parents. Rather than a one point crossover occurring between the parents a multiple point crossover takes place. The two parent strings are divided at random between the two children. If the first parent's feature is assigned to the first child the second parent's feature is assigned to the second child. The probability that the first child has the first parent's feature is the probability of crossover. After generating all children each child has each of its features subjected to the possibility of mutation. Since a feature is a number representing a delay/tap pattern combination in a bucket a random number representing a new delay/tap pattern combination is generated if mutation occurs. The probability of selecting a feature during mutation is inversely proportional to the number of features in a bucket. Once the desired number of children have been produced the fittest are selected as suitable parents for the next generation. The same two sets of data were used to assess the performance of this genetic algorithm as had been used for simulated annealing.

The data used to evaluate the performance of the GA was the same as has been specified for testing of the SA above.

5.6 Results

The following plots demonstrate the performance of simulated annealing and the genetic algorithm's ability to seek the lowest cost function and thus the best PRBS tap combinations for the data. Two data sets were formed with which to evaluate the performance of the simulated annealing algorithm and the genetic algorithm. The sets of data were 31 buckets of delays and associated tap patterns, where $\delta = 10000$ and $2 \leq F \leq 5$. The first set of data consisted of all real tap combinations and associated delays within each tap/delay bucket. This data has an unknown global minima which the above algorithmic techniques are to seek. The second set of data has an artificial global minima created by setting an artificial tap combination in each bucket to all zeros and the delay difference to zero, this pattern would never occur in practice. The aim of this known, forced, global minima was to ascertain the ability of the algorithms in finding this known global minima.

5.6.1 Simulated Annealing

It has previously been explained that simulated annealing has a probability that it will climb out of a minima to a configuration with a higher cost function penalty. Since this higher costing configuration becomes the new working configuration it will not represent the best configuration found by the algorithm. The following result plots display the cost of the best configuration found so far, not the configuration being annealed at that point.

Figure 5.11 and Figure 5.12 show the performance of Scheme 1 and Scheme 2, §5.4, respectively for the first data set with an unknown global minima. It can be seen that Scheme 1, which perturbs a single m-sequence between each cost calculation, descends faster and to a configuration with a lower cost than Scheme 2 which perturbs more m-sequences between each calculation.

Figure 5.13 and Figure 5.14 display the ability of both schemes to find the artificial global minima introduced into the second data set. Again the first annealing scheme out performs the second. Scheme 1 does in fact find the artificial global minima of tap combinations which are all zeros with zero delay difference.

These results demonstrate that simulated annealing is able to find an improved system configuration by means of perturbations of the existing system configuration. Simulated annealing can even find a global minima in a non-exhaustive search of system configurations, the success of this will depend on how striking the global minima is compared with local minima, the case tested here was perhaps over emphasised. However, the speed with which improved configurations are found and how significantly they are an improvement over an initial random configuration depends upon the format of the simulated annealing algorithm. Possible causes for the poor performance of the second scheme relative to the first are that too much heat existed within the system and so it could not settle into an appropriate configuration. Another cause is that it was cooled too rapidly and became frozen into a poor configuration. No attempt was made to find the optimal parameters for each scheme rather to find adequate working parameters.

5.6.2 Genetic Algorithm

Genetic algorithms are stated to be fairly robust to parameter variation, particularly with respect to the crossover rate. To verify this fact the effects of varying the crossover rate and mutation rate were evaluated when the genetic algorithm was applied to the first data set which has an unknown global minima. The ratio of parents:children was fixed for the trials.

For 10 parents and 20 children Figure 5.15 and Figure 5.16 show the effect of varying the crossover rate. The mutation rate was set at 3% or $P_m = 0.03$ which is in the range which texts, [98], recommend. This mutation rate is sufficient to introduce new characteristics into the evolutionary process, but not too large so that the genetic algorithm degenerates into a random search. It can be seen that for this instantiation of a genetic algorithm the rates of cost reduction are very similar as the crossover rate is varied.

For 10 parents, 20 children the mutation rate was varied, Figure 5.17. The crossover rate was fixed at $P_c = 0.5$, since the algorithm has shown to be relatively robust to this parameter its exact value is not too important providing it is constant for all trials. The amount of variation of mutation rate was small but it can be seen that given this fact the algorithm is robust to changes. It was found that if the mutation rate was very low few new features are introduced into the search space and a search of parameter orderings only occurs caused by the crossover, an unsatisfactory reduction in cost function resulted. Likewise if the mutation rate was too large the crossover had little effect since the strings became randomised by the excessive mutation rate.

With the crossover and mutation rate fixed at $P_c = 0.5$ and $P_m = 0.03$ the ratio of parents:children was varied, Figure 5.18 and Figure 5.19. For the genetic algorithm to operate the number of children must be greater than the number of parents since the next set of parents is selected from the present set of children. If the number of parents was greater than the number of children some children would need to be duplicated to form a complete parent group. With the number of parents fixed at 20 and the number of children varied little variation occurs in the rate of cost reduction. Where there is a small group of parents the number of children has little effect since the fittest parents will be the most likely to breed children. Although the pool of children for the next parent generation may be varied in size all children will be of similar capabilities whether this group is large or small. With the number of children fixed and the quantity of parents varied differences in performance can be seen. Poorest performance occurs with a large number of parents and a large number of children. Part of the genetic algorithm is to select the fittest children, thus if a large number of present children are selected to become parents singling out the fittest will not be effective and a strong group of parents will not be formed.

Finally, to test the ability of the genetic algorithm at finding a known global minima in a large search space the second data set was operated upon by the genetic algorithm. Figure 5.20 shows the performance of the algorithm when the ratio of parents:children is varied with the number of children fixed at 50, $P_C = 0.5$ and $P_m = 0.03$. The same effect for the variation in the number of parents is exhibited as for the first data set, that is that for less parents a faster reduction in cost function occurs. Although the known global minima of zero cost is not found within the number of configurations inspected by the genetic algorithm it has certainly got very close. Given more time it would probably cover the remaining reduction.

Comparing the simulated annealing and genetic algorithm results it must be pointed out that 100 more configurations were inspected by the simulated annealing algorithm schemes than by the genetic algorithm. Within a given time, number of configurations inspected, the genetic algorithm outperforms the simulated annealing for reducing the value of the cost function and therefore in finding good tap pattern combinations for multiple PRBS. The smoother curves for genetic algorithms are achieved by averaging several trails with the same parameter set. This was possible due to the faster operation of the genetic algorithm over that of simulated annealing.

5.7 Conclusions

With the aim of being able to encode deterministic information into a stochastic pulse rate signal for manipulation by the processes of §4 an examination of random number generators both in hardware and software has been made. It can be seen that the techniques available are, many and various. One method in particular has been highlighted which can built easily in hardware or modelled in software, the PRBS generator. The PRBS generator consists of an LFSR with an appropriately selected **XOR** feedback circuit which performs modulo two arithmetic. If the feedback combination is correctly chosen an m-sequence is produced with the shift register passing through all its possible states except the all zero state.

Methods for generating delayed variants of the fundamental sequence have been discussed with a view to forming multiple PRBSs from a single generator. The problems of uneven loading upon the LFSR which may be caused by several delayed sequences created from a single PRBS has been drawn attention to before the description of a solution by Alspector which has been implemented for practical use. Alspector left open the question of searching the solution space for an optimum result. To close this gap two methods of combinational optimisation have been experimented with, simulated annealing and genetic algorithms. Both of these techniques utilise stochastic operators.

Simulated annealing and genetic algorithms have both been found worthwhile implementations for the combinational optimisation of PRBS delay tap selection. Two formats of the simulated annealing scheme were tested which can be equated to the amount of energy in the system and the rate of cooling. A difference in performance between the two simulated annealing schemes was noted, thus the exact implementation of simulated annealing to a particular problem is significant. The simulated annealing approach has been found to be several orders of magnitude slower for this problem than the genetic algorithm approach. Intuitively this result is not really surprising in that, although both algorithms involve probabilistic processes, the annealing process does not generate as broad a search space as the genetic algorithm. The genetic algorithm has also been found to satisfy its claim to robustness in the adjustment of some of its main parameters, eg. crossover rate, but more sensitive to other parameters, eg. the number of parents. For this combinational optimisation problem genetic algorithms appear to be the better individual algorithm of the two inspected. Considering the implementation process, simulated annealing is more complicated with the concept of agitating the system, whereas the genetic algorithm involves simply manipulating the string through crossover and mutation.

Simulated annealing and genetic algorithms are not the only optimisation approaches which can be applied, Very Fast Simulated Re-annealing, VFSR, as developed by Ingber and Rosen, [99, 100], is another candidate but this has not been experimented with. Alternatively a hybrid technique drawing on features of both simulated annealing and genetic algorithms could be developed.

All component parts for an artificial neuron operating by the use of stochastic pulse rate computation can now be seen to exist. In the following chapters an actual hardware design is described, implemented and tested before consideration of a suitable training paradigm which may be overlayed onto the fabricated hardware.

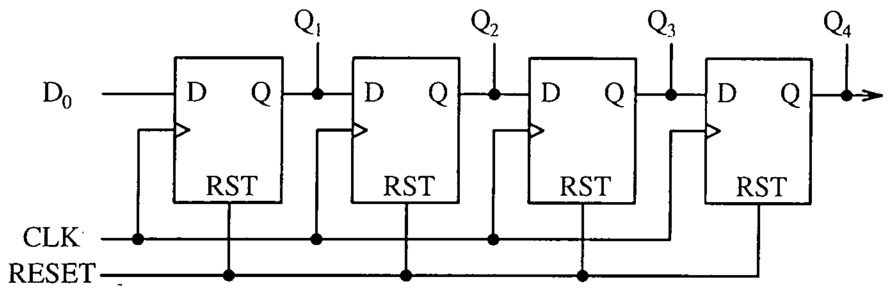


Figure 5.1: Format of a shift register. *The output,* **Q**, *of a given D-type flip-flop stage in the shift register feeds the input,* **D**, *of the following stage.*

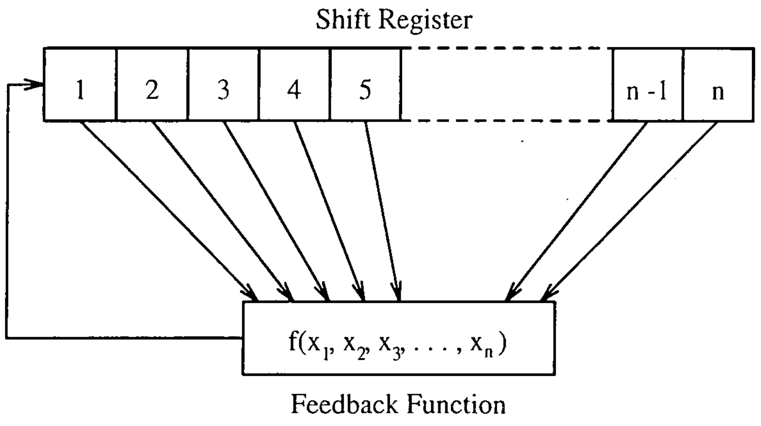


Figure 5.2: Linear feedback shift register, LFSR, configuration. *The input to the first stage of the LFSR is a combination of the outputs from all stages of the shift register.*

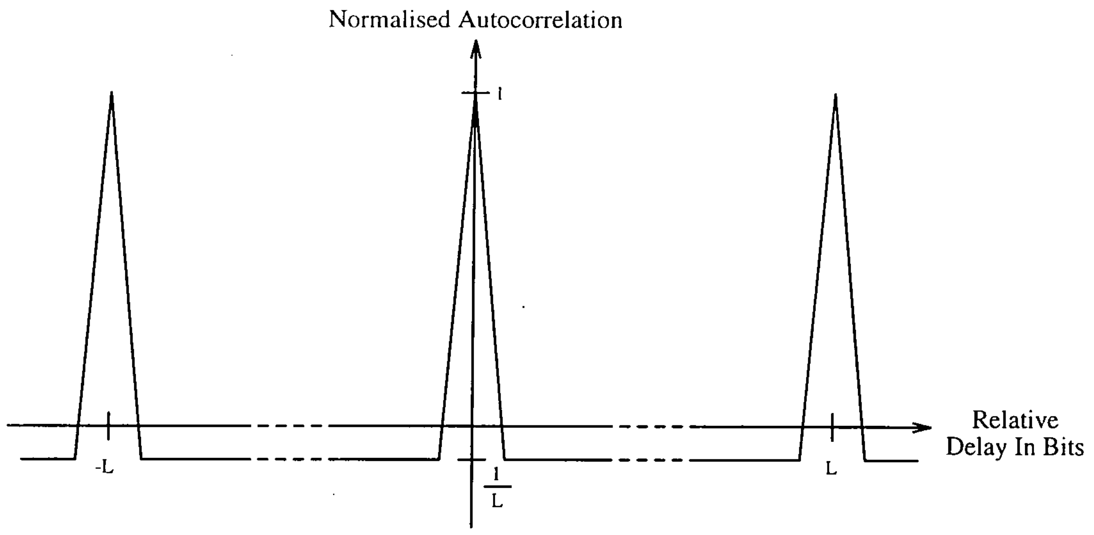


Figure 5.3: Autocorrelation for a PRBS. *The correlation for all except synchronous sequences of the PRBS are negligible.*

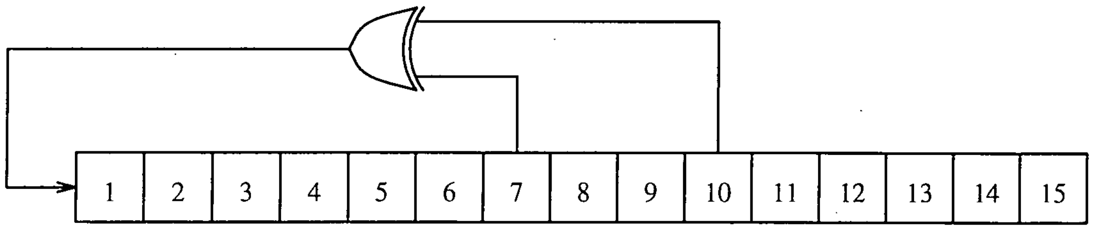


Figure 5.4: Extended PRBS generator. *Delayed versions of a sequence can be obtained by taking outputs from the extensions to the shift register, stages 11-15.*

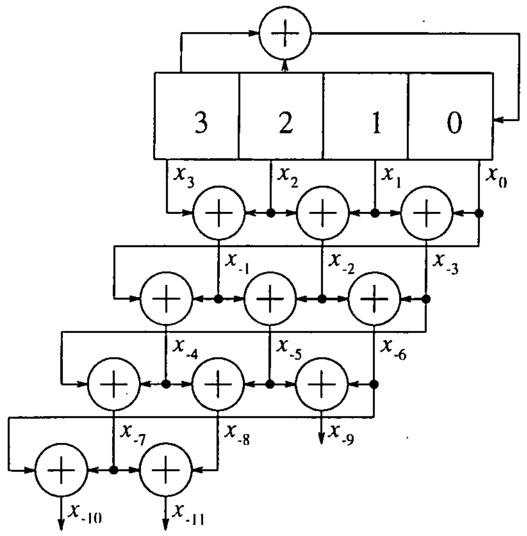


Figure 5.5: Generation of delayed PRBS as illustrated by Tsao. *Modulo two arithmetic and the shift-and-add property of an m-sequence is used to generate delayed versions of a sequence.*

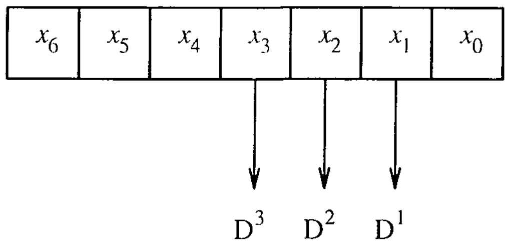


Figure 5.6: Delay variance by moving tap position. *A difference in delay can be obtained by adjusting a tap position up or down the shift register.*

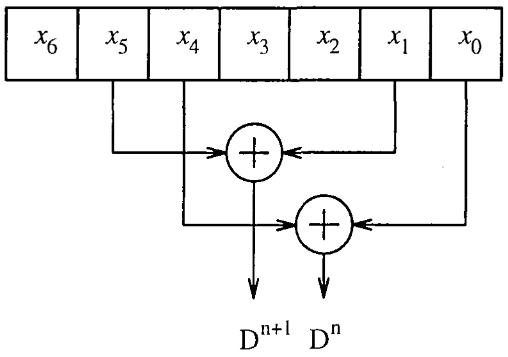


Figure 5.7: Delay variance by moving a set of tap positions. *By extension of the principle illustrated in Figure 5.6 delays generated by tap combinations can be shifted by moving the complete tap combination up or down the shift register.*

1	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1

Figure 5.8: Example of essential taps. *For essential tap patterns the LSB is always unity, near delays are obtained by shifting the tap pattern along the shift register.*

	8	4	2	1	
Direction of shift $\longrightarrow$					
Before Advance	1	0	0	0	$\equiv 8$
After One Advance	X	1	0	0	≥ 4
After Two Advances	X	X	1	0	≥ 2

Figure 5.9: Example of correlation between random numbers formed from successive bits. *It is necessary to advance a shift register by the number of bits it contains to prevent this correlation being exhibited.*

Before Crossover

1	0	0	1	1	1 |	0	1	0	1
1	0	1	0	1	0 |	1	0	1	1

After Crossover

1	0	0	1	1	1 |	1	0	1	1
1	0	1	0	1	0 |	0	1	0	1

Figure 5.10: Illustration of One Point Crossover with Two Strings. *A common point is selected in two strings and the string components are changed at this point.*

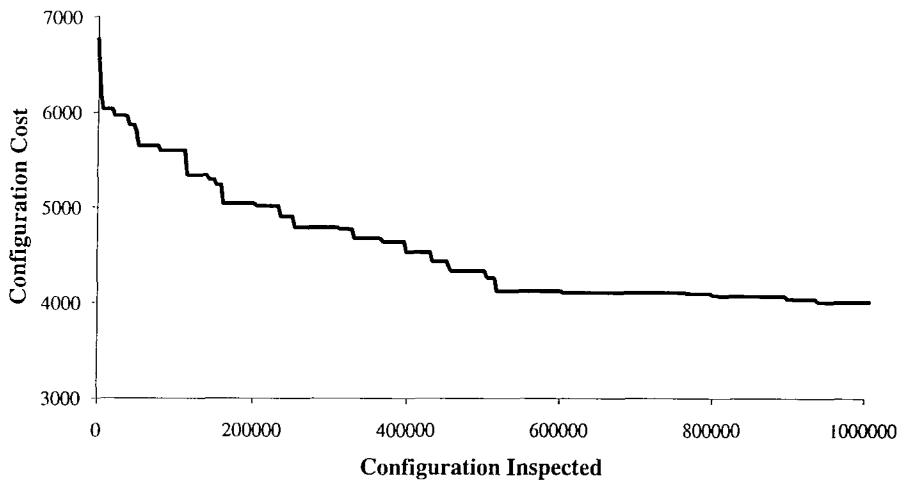


Figure 5.11: Simulated Annealing Scheme 1: Unknown Global Minima. *The cost of tap pattern configurations steadily reduces until 600000 have been inspected at which point the energy minimisation levels off.*

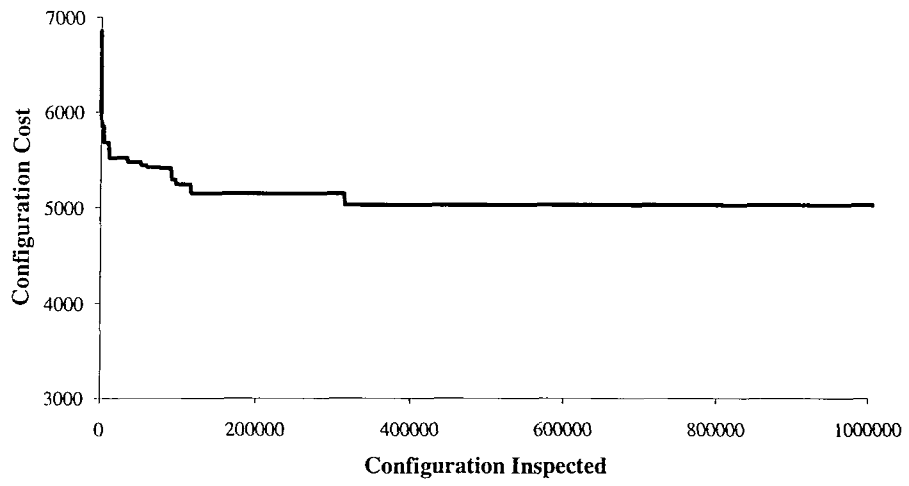


Figure 5.12: Simulated Annealing Scheme 2: Unknown Global Minima. *The cost of tap patterns used decreases but does not reach as low a final configuration and reaches a plateau sooner.*

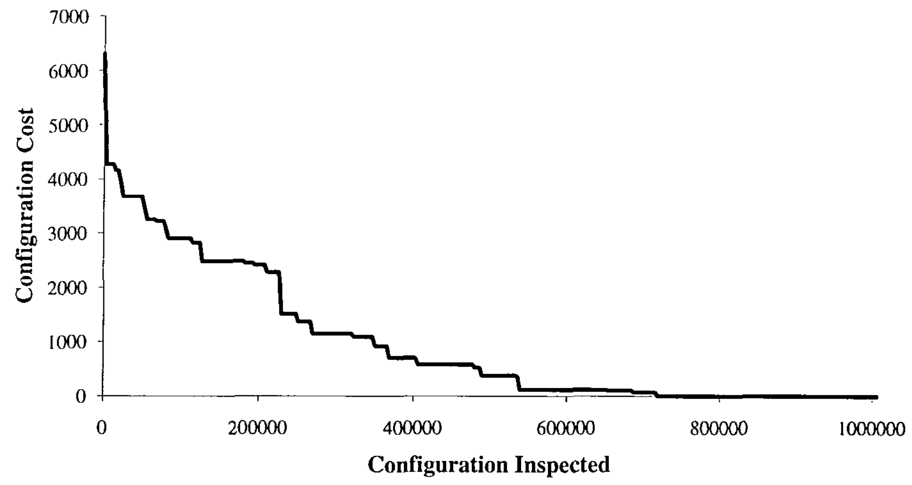


Figure 5.13: Simulated Annealing Scheme 1: Known Global Minima. *This simulated annealing scheme has been able to find the global minima within the search time allocated.*

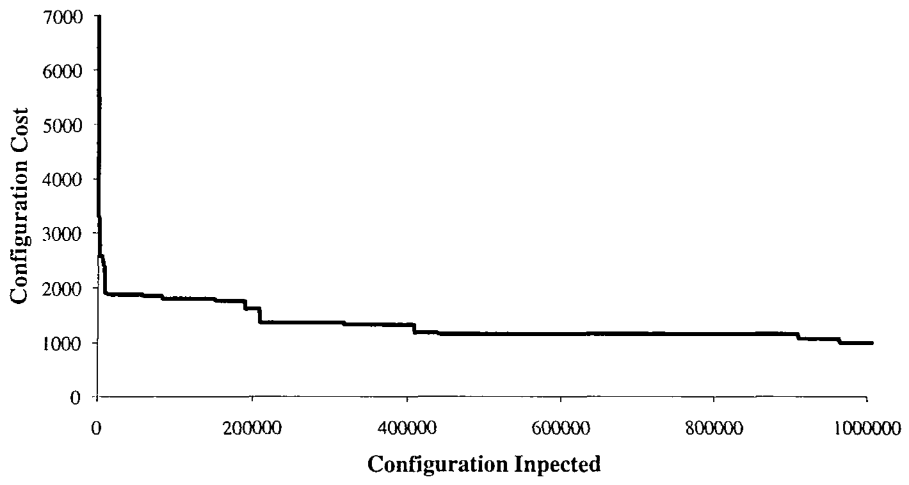


Figure 5.14: Simulated Annealing Scheme 2: Known Global Minima. *This simulated annealing scheme has been unable to find the global minima within the search time allocated, but has tended towards a plateau. Compare this to the alternate scheme Figure 5.13.*

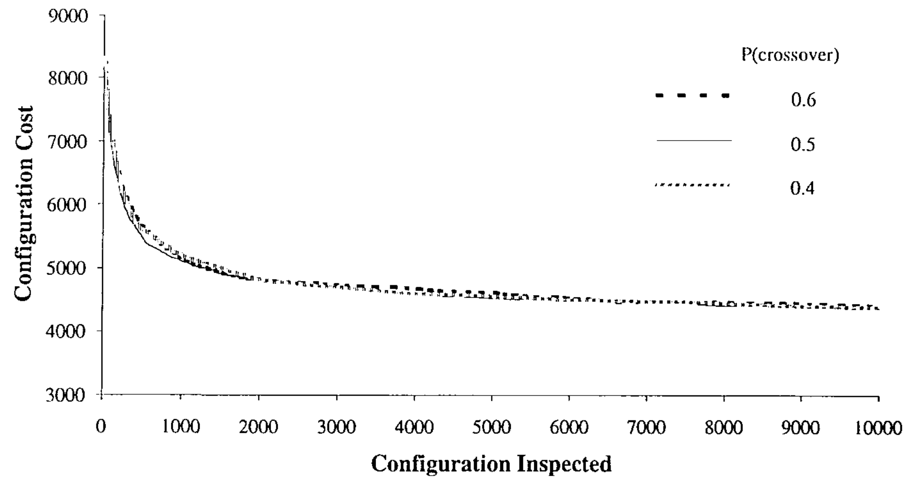


Figure 5.15: Genetic Algorithm: Unknown Global Minima: Varying Crossover Rate. *The genetic algorithm shows little variance in performance for small adjustments in crossover rate. It has reached a comparable minima to that of Scheme 1 for simulated annealing, Figure 5.11.*

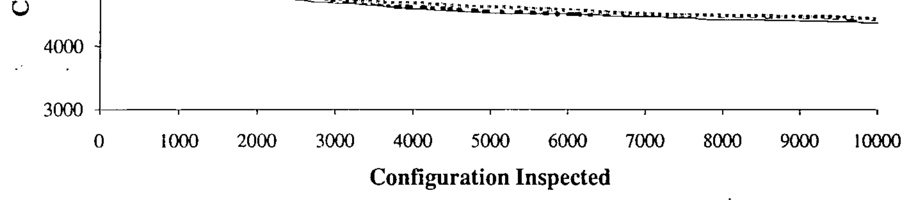


Figure 5.16: Genetic Algorithm: Unknown Global Minima: Varying Crossover Rate. *The genetic algorithm shows little variance in performance for large adjustments in crossover rate, the system is robust for changes in crossover rate. It has reached a comparable minima to that of Scheme 1 for simulated annealing, Figure 5.11.*

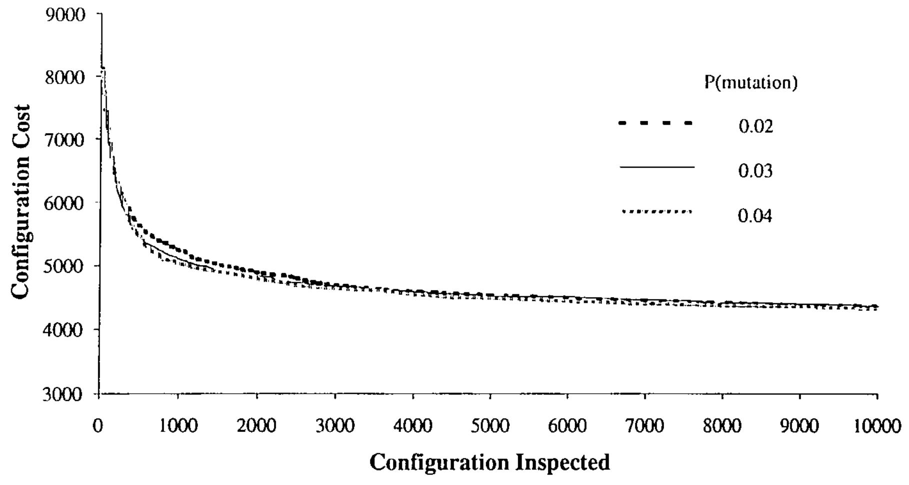


Figure 5.17: Genetic Algorithm: Unknown Global Minima: Varying Mutation Rate. *The genetic algorithm shows little variance in performance for small adjustments in mutation rate. It has reached a comparable minima to that of Scheme 1 for simulated annealing, Figure 5.11.*

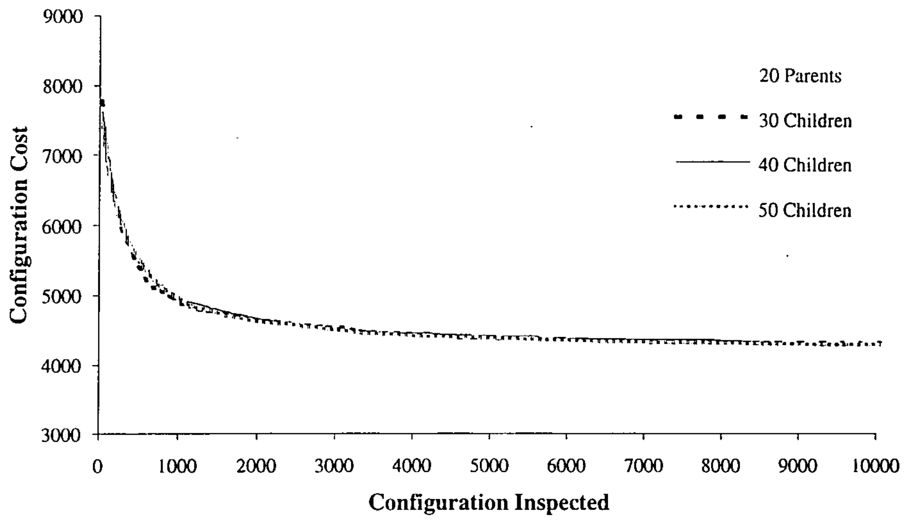


Figure 5.18: Genetic Algorithm: Unknown Global Minima: Varying Parents:Children Ratio. *The genetic algorithm shows little variance in performance for adjustments to the number of children. It has reached a comparable minima to that of Scheme 1 for simulated annealing, Figure 5.11.*

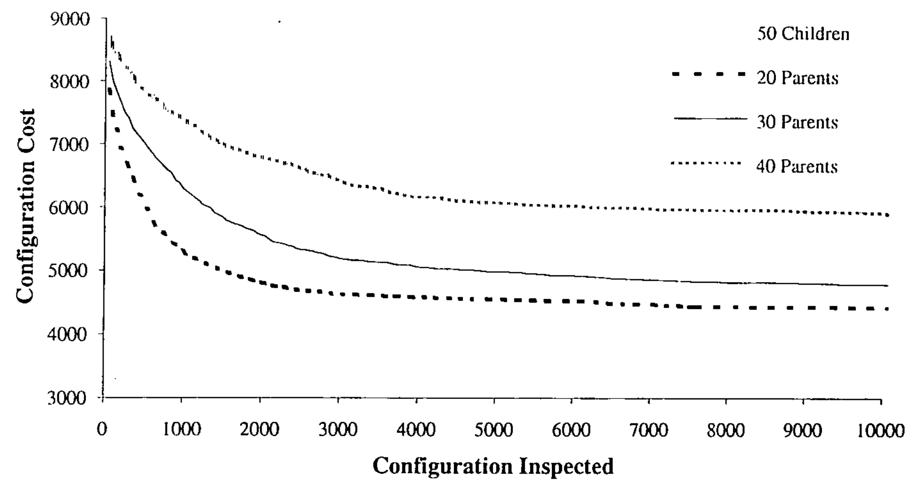


Figure 5.19: Genetic Algorithm: Unknown Global Minima: Varying Parents:Children Ratio. *The genetic algorithm shows quite a degree of variance in performance for adjustments to the number of parents. It is not as robust to adjustments in this parameter.*

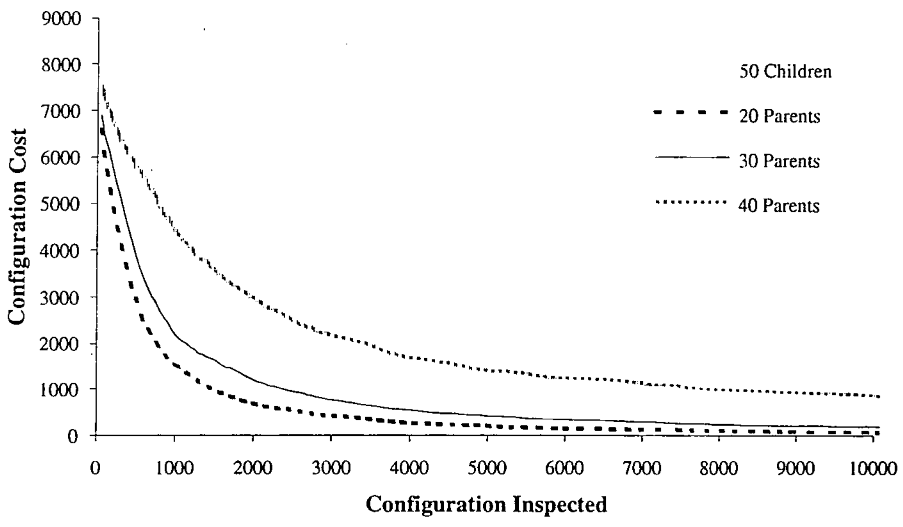


Figure 5.20: Genetic Algorithm: Known Global Minima: Varying Parents:Children Ratio. *As above genetic algorithm shows quite a degree of variance in performance for adjustments to the number of parents. It has tended towards the known global minima quicker for small number of parents.*

Chapter 6

An Artificial Neuron VLSI Design and Implementation

In preceding chapters of this thesis theoretical considerations have been made regarding stochastic pulse rate computation §4 and the random number generation system to be used in such an environment §5. These studies were undertaken with the aim of designing and constructing an ANN operating by the use of stochastic pulse rate encoded signals. An individual neuron must first be designed using these techniques before a whole network may be built. From §3 it can be seen that Banzhaf [57], Kondo *et al* [60], Van Den Bout [55, 56] and Tomlinson [42] have already put forward designs for neurons and ANN. However, these designs either do not operate entirely in the stochastic pulse rate domain, rely upon inexact calculations or are for a particular NN architecture. The design put forward here is for a neuron operating using SLB signals and with all processing performed within the digital domain.

Following an overview of the basic requirements for the neuron architecture to be designed a brief description is made of the design and implementation routes available within the School of Engineering, University of Durham and the reasons for selecting the ASIC design package Solo 1400. The next section of this chapter is concerned with the design and development of working sub-circuits before they are connected together to form a working neuron. Finally, there follows a description of the test system used and the tests applied to a fabricated neuron device.

6.1 Neuron Overview

For a neuron to be practically realised in hardware several factors must be examined. Firstly, the method of computation and communication must be considered. This has been decided upon as being stochastic pulse rate encoded signals, but should this be unipolar, bipolar, single line, dual line, linear or non-linear? Since many of the constituent parts of a neuron (multipliers and summers) have not been considered for non-linear encoding strategies the design must be a linear one. Linear dual line circuits tend to be larger than their single line counterparts, using a single line scheme will lead to smaller circuits. In addition the routeing of signals between component parts will be easier for the single line rather than for the dual line case. Overall the signal computation should be bipolar although it may be found that unipolar signals are more appropriate for some applications within the neuron.

Secondly, the size of the neuron must be considered, what fan-in should it have ie. how many inputs will there be? This will be governed ostensibly by the task the NN has to perform in which the neuron is placed. If the neuron has excess inputs it is possible to set unused inputs to zero so they do not contribute to the processing, whereas if there are insufficient neuron inputs additional inputs cannot be added. Too many inputs will lead to a large neuron which may prove unwieldy in this proof of principle exercise. For these reasons a fan-in of 16 was selected. From an estimate of the circuitry size and complexity to implement the design it should not prove too large to fabricate and test. In addition the design is not too small that a computationally useful task cannot be performed.

Thirdly, the technology with which the neuron is to be built must be considered. Whether to use discrete ICs or VLSI design tools? Whether it will be TTL or CMOS? If a VLSI design is implemented what level of design is necessary, eg. full custom, standard cell? These questions about realisation are considered more completely in the following section.

To summarise a general artificial neuron using SLB stochastic pulse rate encoded signals with 16 inputs is to be built. The basic layout of neuron is as per Figure 2.2 a sum of weighted inputs passed through an activation function, a sigmoid transform in this design. The performance of the neuron can be adjusted by varying the weights and so these weights must be programmable. If the neuron is to be used in a circuit which learns and adapts on-line then the weights must be able to be varied as the neuron operates. The block diagram for the neuron is Figure 6.1.

6.2 Design Tools

Within the School of Engineering several options exist for the fabrication and test of an artificial neuron. The three options considered are the construction from discrete TTL or CMOS components, design/simulation/layout via the Solo 1400 CAD package and finally design/layout/simulation using a combination of ChipWise, SPICE and System HILO 4 CAD tools. Each of these three options offers varying degrees of sophistication, adaptability, testability, expense and lead time. Each option will now be described in turn before reviewing slightly more deeply the selected option.

Construction using discrete TTL or CMOS components offers the most flexibility in the built hardware. Standard components may be used to perform specific tasks and circuits easily adapted if a design alters. This flexibility is also the weakness in that construction becomes prone to errors. For a single neuron device this may be the best option to take, but if many neurons are to be built the job becomes highly repetitive with the increased liability of errors. There is little delay between design and test as the circuit exists from the outset. The discrete nature also means that there are more points at which a circuit can be externally monitored to verify performance.

The second option is the use of the Solo 1400 ASIC design package. This tool allows the design, simulation, circuit layout and packaging to be accomplished in a unified environment before dispatching the design to be fabricated by a third party. Solo 1400 makes use of fully characterised standard cells of devices and circuits in 1.2μm, 1.5μm and 2.0μm CMOS technology which can be interconnected to form higher level functional circuits. Libraries of intermediate circuits, eg. counters and registers, are available to speed the prototyping phase. Once the neuron design is complete many can be fabricated at the same time. It is not feasible to make changes to a design once it has been fabricated, thorough design and simulation is therefore necessary.

The third and final option is also an integrated circuit approach. The aim would be to utilise a combination of ChipWise, SPICE and System HILO 4 to produce a full custom designed neuron. It would be necessary to design the individual logic gates through the more complex sub-circuits to the final complete neuron. In effect a personal library of components must be built and tested. The components gates can be simulated and characterised using the SPICE circuit simulation tool which could be used to extract timings and drive capability information for example. The extracted parameters would be inserted into a circuit description within System HILO 4 to allow simulation of the functionality of connected circuits over a period of time as the circuits run. Circuit layout and routeing of a design must all be accomplished manually. Once a design had been completed it would need to be fabricated and packaged by a third party. This option has potentially the most sophisticated result but requires a prohibitive quantity of work to be undertaken.

In fabricating a neuron a balance has to be struck between design flexibility and adjustment, ease of testing, level of integration, sophistication of design and repeatability of fabrication. Each of the three above systems has strengths in some but not all areas. Solo 1400 with its unified environment offers the best compromise since this will allow the production of ASICs with their high level of integration and a structured format of design simulation and test. Through the use of standard cells the individual design and characterisation of many circuit components has already been accomplished allowing the structuring of the design with relative ease. The integrated simulation should enable the highest probability of a functioning neuron to be designed.

6.2.1 The Solo 1400 Program Suite

Solo 1400 consists of several separate programs instantiated from within a Solo 1400 environment shell, in this case running under the X11 windowing system upon a Unix workstation. The programs used can be classified into five general groups,

Design entry using **draft** or an ordinary text editor.

Circuit compilation with the `model` language compiler **model**.

Simulation and test with the waveform compiler **wdl**, simulator **mads** and output inspection utility **wave**.

Layout and encapsulation of the design using **place**, **gate**, **pinout**, **route**, **draw**, **artview** and **package** utilities.

Design management using **audit**, **padaudit** and **shipdes**.

This is not a full list of the extensive Solo 1400 programs complete details of which can be found in the reference manuals [101].

Design Entry

Two systems were used to enter a design, the first being the schematic entry utility **draft**, the second an ordinary text editor with which to write a circuit description using the `model` hardware description language, HDL.

With the **draft** tool a GUI interface is used to select, place and connect components together. Libraries of pre-designed circuits, either standard or user written, can be called upon to be added to the schematic. The resulting circuit may then be encapsulated within a symbol as a new component for use in a higher level circuit. A hierarchy of building blocks is constructed for a design such that at the highest level all that may be seen is a number of interconnected black boxes with input/output pads attached. The resulting output from **draft** is a compilable text file of `model` code.

Textual entry of a circuit design uses the `model` HDL. The structure of the language is simple and clean, it is not unlike writing a conventional software program. With the experience of using **draft** a hierarchy of circuits can be written in either library files `<library>.inc` or actual compilable circuit files `<circuit>.mod`.

Both systems had their place in the design process. Initially schematic entry provides good visual feedback of the design of the circuit but it is much slower for design entry as the size of a circuit grows. Due to the name checking facility of **draft** circuit interconnection can be a problem as names on buses, wires may not agree even though such a connection is valid. It was found that often a base design could be produced using **draft** and the `model` code produced extracted and incorporated into a textual library where minor variations were made for specific needs. Conversely text based designs would be bound into a schematic so that pads could be connected and the sub-circuit exported for standalone simulation and testing.

Circuit Compilation

Following the entry of the circuit(s) and the formation of a `model` file the code is compiled using the **model** utility. **model** expands a `<circuit>.mod` file to a `<circuit>.mdl` file which is compiled into a `<circuit>.idl` file for the simulation of the design. Checks upon the design for integrity are made with errors and anomalies adequately reported.

Circuit Simulation and Test

Having produced a valid circuit design it is necessary to verify its operation and performance. Solo 1400 offers several tools for this task, the main one used was **mads** (Multi-level Analogue and Digital Simulator) together with the **wave** utility for displaying the output.

mads takes as its input a `<circuit>.wdl` file which describes how the circuit inputs are to be driven, outputs of the circuit and specified monitor points within the design are logged as the simulation progresses. The `<circuit>.wdl` is a text file written in WDL (Waveform Description Language) which is very similar to `C` but with notable differences, eg. no array handling. A well written exercise routine greatly aids in the verification of a design and in debugging should this be necessary. It is not possible to specify what an output or monitor point should be at any given time, this must be deduced from the **wave** output and checked manually. These same test files for a circuit can/are used at a later stage in the design process after the circuit layout and encapsulation when a greater knowledge of the timing considerations are available and when testing with actual device parameters occurs accounting for propagation delays, device loadings, tolerances etc.

Circuit Layout and Encapsulation

Given that a circuit operates as expected the next stage is to lay the design out on silicon and if desired encapsulate within an appropriate package. Most of this process is mechanical but user intervention is possible to fine tune parameters if desired. Normally the default performance will prove satisfactory.

The first step is the execution of **place**, a utility which resolves the design hierarchy into basic logic gates. The resolved hierarchy as implemented by a series of stages is drawn out into a long line and then set out in a regular structure of rows and columns by repeatedly folding the long line of stages back and forth to form an approximate square format.

The **gate** utility constructs each actual gate upon the output from the **place** utility.

The next stage of the circuit layout is the routeing of wires between gates and out to the pads. User intervention is required to organise the pads on the die to a desired format, the **pinout** utility provides a GUI based system for performing this operation. Included in **pinout** is the ability to select the desired package in which the resulting die will be encapsulated, it was found that for experimental designs the default package selected based upon the die size was satisfactory. Once the pad organisation is complete **route** can be executed which performs the actual placement of interconnections. User intervention for the routeing process will have taken place at the design stage with the specification of time critical signals.

It is possible to inspect the resulting artwork from the placement and routeing using **draw** and **artview**. **draw** translates the final output from **route** into Caltech Interchange Format[1], ie. it generates a `.cif` file. The `.cif` file can be inspected graphically using **artview** which allows the mask design to be viewed at various levels of detail. It is feasible to zoom into areas of the mask and to measure distances between circuit elements.

The final stage is to encapsulate the design into the specified package from the **pinout** phase. This entails placing the bonding wires between the die pads and the pins of the physical package using **package**. **package** is similar to **pinout** in the layout of the GUI and the package type selected should correlate with that selected in **pinout** else it will be necessary to cycle back to the **pinout** stage.

After placement and routeing it is required to return to the simulation and test phase where the simulations can be rerun but with greater detail of device parameters and propagation delays. Simulation runs accounting for maximum, minimum and nominal expected timings should be successfully executed.

Design Management

To aid and maintain consistency of a design through its various stages Solo 1400 has several utilities for automatically generating templates of required files **extract** and of analysing the circuit produced **audit** and **padaudit**. **shipdes** is used to check the integrity of the overall design process, that all the required utilities have been executed successfully in the correct order, that all the test phases have been executed and any special concessions have been agreed with the fabrication institution.

6.3 Artificial Neuron Design

Each of the sub-circuits of the artificial neuron design will be specified before amalgamation into a single neuron unit for simulation and fabrication. A modular approach to design has been adopted since an artificial neuron can then be constructed from tested sub-assemblies with known modes of operation. Many of the sub-circuits are reused, by designing in a modular format new occurrences of a module can be instantiated reducing the risk of

[1]**CIF** Caltech Intermediate Format is a system for describing graphics items, mask layouts, in a machine readable form for use by an output device.

errors and keeping the circuit description to a minimum. For example, in the case of the N pulse divider weight encoders §6.3.5 a basic module was adapted and renamed for each of the required weights. The sub-circuits are now presented either as **draft** printouts or in the form of sample `model` code. Example **wdl** test files are shown together with the associated **wave** output plots.

6.3.1 PRBS Generator

A PRBS generator is required to create the random numbers which are to be used for the encoding of the neuron input weights, the N pulse divider weights and the sigmoidal transform. By the use of Alspector's technique [94] as discussed in §5.3.3 multiple PRBS sequences from a single generator may be formed, actually the same sequence but at different positions in its run length. The total number of sequences required for the neuron is 34, made up of 17 for encoding the neuron input weight values, 16 for encoding the N pulse divider weights and a single sequence for the sigmoidal transform.

A 27 bit PRBS is used, a schematic of which is shown in Figure 6.2 and the `model` code listing for the variable length shift register is shown in Figure 6.3. The appropriate PRBS feedback points were obtained from a table of primitive polynomials [102] which are known to produce maximal length sequences. In order to allow for additional PRBS sequences which may be required the software developed for the implementation of Alspector's technique §5.3.3 was used to find a total of 38 sequences with a minimum of two taps and a maximum five taps used. The delay variation, δdelay, from the nominal was set at 100000. Thus, the nominal spacing between sequences is

$$\frac{\text{Maximal length}}{\text{Number of sequences}} = \frac{2^{27}-1}{38} \approx 3532045$$

and the worst case spacing between sequences will be

$$\text{Nominal space} - 2 \times \delta\text{delay} \equiv 3332045$$

A suitable configuration for the tap off sequence gating was found by the use of the simulated annealing software §5.4. A sample of the `model` file for `prbs27to38` which generates the circuit is given in Figure 6.4.

No simulation of the tap off sequences were made but the basic PRBS generator was exercised using **mads** and the `wdl` file of Figure 6.5. For this `wdl` file the generator is reset such that all the individual elements are 1 and then run for 50 clock cycles at which time it is reset again and run for a second 50 clock cycles, both should produce the same results. It can be seen from the waveform plot of Figure 6.6 how the generator operates for this short period of time and that it is successfully reset at time = 102500.

6.3.2 12-bit Comparator

In the following two sections the storage and encoding for the neuron input weights and the N pulse divider weights will be explained. Central to the transformation from a deterministic value to a stochastic pulse is a circuit for comparing a weight register value W with a random number R which has the same number of bits. If $W > R$ a one is required as an output else a zero is output. For the basic one-bit case the circuit of Figure 6.7 will achieve the objective for arbitrary values of A and B. However, it is required to compare two n-bit numbers. For example, consider two n-bit numbers where n = 3 such that $X = X_3X_2X_1$ and $Y = Y_3Y_2Y_1$. A possible algorithm for comparing these values is

1. Examine the MSBs, X_3 and Y_3

$$\begin{aligned}&\text{if } X_3 > Y_3 \text{ then } X > Y\\&\text{if } X_3 < Y_3 \text{ then } X < Y\\&\text{if } X_3 = Y_3 \text{ then no decision}\end{aligned}$$

2. Examine the next two bits, X_2 and Y_2

$$\begin{aligned}&\text{if } X_2 > Y_2 \text{ and } X_3 = Y_3 \text{ then } X > Y\\&\text{if } X_2 < Y_2 \text{ and } X_3 = Y_3 \text{ then } X < Y\\&\text{if } X_2 = Y_2 \text{ and } X_3 = Y_3 \text{ then no decision}\end{aligned}$$

3. Finally, examine the last two bits X_1 and Y_1

$$\begin{aligned}&\text{if } X_1 > Y_1 \text{ and } X_3 = Y_3,\ X_2 = Y_2 \text{ then } X > Y\\&\text{if } X_1 < Y_1 \text{ and } X_3 = Y_3,\ X_2 = Y_2 \text{ then } X < Y\\&\text{if } X_1 = Y_1 \text{ and } X_3 = Y_3,\ X_2 = Y_2 \text{ then } X = Y\end{aligned}$$

This algorithm could be expanded in logical form as follows where E_p is the equivalence of any individual p bits.

$$\begin{aligned}E_3 &= \bar{X}_3\bar{Y}_3 + X_3Y_3\\E_2 &= \bar{X}_2\bar{Y}_2 + X_2Y_2\\E_1 &= \bar{X}_1\bar{Y}_1 + X_1Y_1\end{aligned}$$

therefore

$$\begin{aligned}X = Y &: E_3E_2E_1\\X > Y &: X_3\bar{Y}_3 + E_3X_2\bar{Y}_2 + E_3E_2X_1\bar{Y}_1\\X < Y &: \bar{X}_3Y_3 + E_3\bar{X}_2Y_2 + E_3E_2\bar{X}_1Y_1\end{aligned}$$

The logic gating even for only this 3-bit case is becoming quite involved. Fortunately there is a more efficient system, in terms of gating, which can be utilised, the iterative comparator.

It can be seen from the explanation of the 3-bit comparator operation above that a pattern of operation is emerging ie. given that no decision has been possible as to which is greater X or Y then compare the next MSBs. In the worst case it is necessary to compare all bits of the two numbers to form a decision. Some logic design books eg. Holdsworth [103] and Roth [104] provide the derivation for the iterative comparator which is illustrated in Figure 6.8 and Figure 6.9. The operation is that with a_0 and b_0, which are equivalent to Z_1 and Z_2 reset to zero, to compare the two bit streams of X and Y a bit at a time starting with the MSB and recycle the result at each clock pulse for each subsequent bit. A valid comparison result may occur before all bits have been compared but it is necessary to wait for the final bit comparison to be certain of the correct result.

The iterative comparator circuit is sequential whereas the original comparator described was made only from combinational logic. It is true to say that the iterative comparator is slower than the combinational comparator at actually testing the two numbers but the combinational comparator has to wait for the full numbers to be formed, probably in shift registers, before the computation can take place. There is thus no time disadvantage to using the iterative comparator in this case but there is a great benefit in terms of the circuit complexity and component count. The length of the numbers that can be compared by this iterative technique is determined by the clocking and reading arrangement not by the fundamental logic design of the comparator.

Figure 6.10 and Figure 6.11 illustrate the `model` code used to generate the iterative comparator. After compiling an encapsulation of this design it was exercised using the `wdl` file of Figure 6.12 the results of which are seen in Figure 6.13. For this simulation three 4-bit comparisons were undertaken $(1011, 1100) \equiv (10, 12)$ starting at time 240, $(1101, 1011) \equiv (13, 11)$ at time 750 and $(0101, 0101) \equiv (5, 5)$ at time 1250. The reset line is taken low before each comparison begins to clear the output latches of any value they may hold. It can be seen that R goes high and T goes high at the points the conditions $X < Y$ and $X > Y$ are detected respectively. Both R and T remain low where the input signals are identical. The extension to 12-bit numbers is achieved by entering 12-bit numbers, MSBs first, into the comparator and increasing the period between the reset pulses.

6.3.3 Counters

Solo 1400 contains several libraries of elements including `firmlib` and `synclib`, within these libraries are more sophisticated circuits eg. multiplexors, n-bit shift registers and counters. For the neuron design two types of counter are required, firstly a basic counter which can be loaded with a specific value from which to start counting, secondly a more sophisticated up/down counter which can also be loaded with a specific value. Only the former exists in the libraries, a synchronous counter. The latter up/down counter will need to be constructed from basic logic gates.

Starting with the basic synchronous counter one is required to count up to the number of bits being compared by an iterative comparator, 12, and then reset both itself and the comparators. Another of similar form but with a count of 80 is required for monitoring of the E-sequence progression in the sigmoidal transform circuit. `model` files for the two counters are in Figure 6.14 and Figure 6.15. The format of the two counters is the same with a synchronous counter at the heart which is reset to all zero either by the count of 12 (80) being reached as detected by the immediate logic gates on its outputs. Alternatively the counter may be reset to zero by an externally applied reset signal. A single low pulse `rstcnt` clocked through a flip-flop when the counter reaches its limit is produced. The `wdl` file and associated waveform plot are shown for only the 12-bit counter in Figure 6.16 and Figure 6.17. The `Probe` commands in the `model` code allows signal lines internal to the circuit to be monitored as well as the external connections which are always monitored.

By probing internal lines spikes can be seen upon `rstcnt` which propagates to `rstes` this is caused by the propagation of signals through the combinational logic on the outputs of the counter. The spike is hidden from the reset input of the counter by the d-type flip-flop and causes no problems.

Moving onto the second type of counter, the up/down counter, a 12-bit variant is required for the storage and adjustment of the input weight values. An up/down counter description does not exist in Solo 1400 so rather than redesigning a fairly common system the 74169 TTL circuit was transcribed and used, Figure 6.18. The 74169 circuit is a 4-bit up/down counter with both a carry-in and a carry-out, it can be loaded with an arbitrary 4-bit value. By cascading three devices a 12-bit up/down counter could be formed. Since an up/down counter is required for a total of 17 input weights two variations on the 4-bit counter were formed one with no carry-in circuitry, Figure 6.19, and one with no carry-out circuitry, Figure 6.20 enabling the 12-bit up/down counter of Figure 6.21 to be generated.

For 12-bits the range of numbers is $0 \rightarrow 4095$ or for symmetrically distributed bipolar values $-2048 \rightarrow +2047$ which is required here. A means for inhibiting the counter movement when reaching either of these limits is required which will also allow the counter to move away from the limit if the opposite direction signal is applied. The final 12-bit up/down counter circuit with limit stops is displayed in Figure 6.22. For exercising and simulating this circuit a `wdl` file was used to verify that any value could be loaded into the counter, that all the crossings from the use of one 4-bit stage to another operated both ascending and descending in both positive and negative halves of the number range and finally that the maximum and minimum limit stops operated satisfactorily. A `wdl` file and associated **wave** plot for the two limit tests are shown in Figure 6.23 and Figure 6.24.

In the simulation, Figure 6.24, the counter is loaded with a value just less than the maximum ie. 0x7FA at time ≈ 2500. With `UD` set HIGH which is equivalent to up the counter can be seen to count up, `cntout(0:11)`. When the counter reaches the maximum value it stops until the count direction is changed time $\approx$ 22000 when it starts to count down. The process is mirrored for checking the minimum value limit stop starting at time $\approx$ 32700 when a value just greater than the minimum is loaded into the counter.

A 5-bit up/down counter is required in the Gaussian random generator of the sigmoidal transform. This is required to have a lower limit of 0 and an upper limit of 80, this counter does not need to deal with negative numbers. Rather than use the two appropriate 4-bit counters and limiting the counting as per the 12-bit version it was decided to extend the principle by which the 4-bit operated to five with no carry-in and no carry-out. The resulting circuit is shown in Figure 6.25 and with the count limiting circuitry added in Figure 6.26.

As per the 12-bit variant the counter was exercised and simulated at being loaded with a valid value, counting up/down and stopping at the two limit points until the direction of count was changed. No figures illustrate the `wdl` file or **wave** output plot.

6.3.4 Input Weight Storage and Encoding

The 12-bit up/down counter with limit stops described in the previous section §6.3.3 forms the basis for the input weight storage and encoder circuit a diagram of which is shown in Figure 6.27. In this circuit the 12-bit weight value, $-2048 \leq W \leq 2047$, is held in the up/down counter. It can be adjusted either by loading a new value explicitly or by counting up or down thus allowing the weight to change as the artificial neuron operates. Every 12 clock pulses the value in the counter is transferred to a shift register. In this transfer the MSB is inverted, the effect of this inversion is to translate the number range up by 2048. The new 12-bit number is compared a bit at a time with one of the 38 PRBS sequences from the PRBS generator §6.3.1 by a 12-bit iterative comparator §6.3.2. The result of this comparison is latched out after the 12th bit has been compared at which point the new weight value is transferred into the comparator register and the process repeats itself.

Since the up/down counter receives every clock pulse its value will constantly be counting up or down in this arrangement. In order to maintain a stable value to be encoded it is necessary that the average number of counts up is equal to the average number of counts down. A stochastic pulse sequence of value 0.5 should thus be fed to the Up/Down input. The value of 0.5 corresponds to zero in a SLB stochastic computation scheme.

Originally this part was designed using the **draft** schematic editor, but after the basic layout had been produced the `model` part description was extracted, edited and debugged resulting in the final description of Figure 6.28. The major components of Figure 6.27 can be identified as follows, Up/Down Counter $\rightarrow$ `ud12bitst`, Comparator Register $\rightarrow$ `es2sreg ps` and the 12-Bit Iterative Comparator $\rightarrow$ `comp_iter`.

A total of 17 of these circuits are required which could mean many connection points to the outside world from the ASIC if the 12 weight input lines and the 12 weight output lines are all separate. This is resolved by using bi-directional pads for the weight input/output immediately halving the number of connections at the expense of a little control logic. Secondly by developing a simple address decoder/demultiplexor to select which input weight is required for writing to or reading from, together with a multiplexor for selecting the appropriate lines if a weight value is to be read out the number of connections can be reduced to one set of 12. The model descriptions of Figure 6.29 and Figure 6.30 illustrates the address decoder and multiplexor implementations respectively.

Appropriately combining 17 input weight encoders, address decoder, 12 multiplexors with the necessary drive buffering a unified input weight encoding block can be formed. This is not illustrated.

Simulation and verification was conducted upon the component parts of the input weight system before utilising the entire system. Taking first the input weight encoder itself it was loaded with three values , $0 \equiv$ 0x000, $+1024 \equiv$ 0x400 and $-1024 \equiv$ 0xc00, which for a 12 bit range, $-2048 \leq x \leq +2047$, should result in a stochastic pulse stream of value 0.5, 0.75 and 0.25 respectively. This is borne out by the **wave** plot of Figure 6.31 where `T` is the output pulse stream. The `LD/EN` pulses can clearly be seen with the corresponding changes in `IN(0:11)` time ≈ 0, 2400000 and 4800000. `UD` and `CLK`, the up/down and clock signals, appear as solid bands since on the scale of the plot they are varying too quickly to be able to observe individual movements. Testing the address decoder is trivial with five input address lines and 17 output select lines. By counting up through the binary codes `addr(0:4)` inputs Figure 6.32 demonstrates that each of the select lines `select(0:16)` is chosen correctly.

6.3.5 N Pulse Divider Weight Encoder

For the N pulse divider which will be used to bias all 17 weighted input lines it is necessary to generate 17 stochastic pulse streams of value $\frac{1}{17}$ for which is needed a series of pulse streams of $\frac{1}{17}$, $\frac{1}{16}$, ...etc. By encoding unipolar values of $\frac{1}{17} \times$ FS, $\frac{1}{16} \times$ FS, ...the stochastic pulse streams can be formed. FS is the full scale value. The basic encoding circuit is illustrated in Figure 6.33 and is similar to the input weight circuit encoder of Figure 6.27 but slightly simpler since there is no up/down counter to be included. As the system is used to encode a constant value the inputs to the shift register are tied to the power rail or to ground so that upon reset it reloads its unique value for encoding. Due to the uniqueness of the load value a separate description must be produced for each encoder. Figure 6.34 displays the `model` code for this encoder for the value of $\frac{1}{10}$ while Table 6.1 is a table of values bias register contents.

The simulation of this circuit follows the same format of the input weight encoding simulation of the previous section with the output stochastic pulse streams of the appropriate value. This will actually be illustrated in the full simulation of the N pulse divider when the output of these fixed value unipolar encoders will be probed.

	Register Contents	
Bias Register	Decimal	Hexadecimal
$\frac{1}{17}$	241	0x0F1
$\frac{1}{16}$	256	0x100
$\frac{1}{15}$	273	0x111
$\frac{1}{14}$	293	0x125
$\frac{1}{13}$	315	0x136
$\frac{1}{12}$	341	0x155
$\frac{1}{11}$	372	0x174
$\frac{1}{10}$	410	0x19A
$\frac{1}{9}$	455	0x1C7
$\frac{1}{8}$	512	0x200
$\frac{1}{7}$	585	0x249
$\frac{1}{6}$	683	0x2A6
$\frac{1}{5}$	819	0x333
$\frac{1}{4}$	1024	0x400
$\frac{1}{3}$	1365	0x555
$\frac{1}{2}$	2048	0x800

Table 6.1: N Bias Register Contents

6.3.6 N Pulse Divider

In the proposal of §4.4.1 for an N input adder circuit an extendable circuit for generating N pulse streams of $\frac{1}{N}$ is shown, Figure 4.12. This circuit will now be modelled using Solo 1400. It will be noticed that a basic cell of two **AND** gates and an inverter exists which is repeated in a ladder structure. This basic block `divide_cell` is realised as an individual element in the `model` code Figure 6.35 which allows an arbitrary sized N pulse divider to be specified using the parametrised `model` code Figure 6.36 where the `divide_cell` block is repeatedly used.

For the neuron circuit 17 pulse streams of value $\frac{1}{17}$ are required so a simulation using all pulse divider weight encoder circuits was simulated. By probing the output of the weight encoders which are internal to the circuit the operation of all the encoder can be verified at once. The **wave** plots of Figure 6.37 and Figure 6.38 displays all the encoded weights `n(2:17)` are the resultant $\frac{1}{17}$ pulse streams `u(1:16)` respectively, `u(0)` $\equiv$ `n(17)`. Spikes can be seen in the $\frac{1}{17}$ pulse streams but due to latching of data in later parts of the neuron these do not cause problems.

6.3.7 Multipliers, Gating and Summation

These circuits are as discussed in the review of stochastic computation techniques §4 and are trivial. For performing the multiplication between the input value and its associated weight a single **XOR** gate is used. To make the circuit definition less error prone a parameterised array of **XOR** gates is specified given by the `model` code of Figure 6.39.

The original N input adder design is used to perform summation of the weighted input signals. In order to gate these values appropriately the 17 lines of output from the N pulse divider `u(:16)` are used to gate the weighted input signals using a parameterised array of **AND** gates as per the **XOR** gate case above Figure 6.40.

Finally, the signals can be summed using an **OR** gate without the fear of losing information due to the coincidence of input pulses or performing inexact computation. The original choice was to use a tree structure of two and three input **OR** gates. Fortunately Solo 1400 contains a built in parameterised **OR** gate circuit which can take N inputs, in this case N= 17. This component will lead to a more efficient and compact multiple input **OR** gate. This is not illustrated.

6.3.8 Sigmoid Transform

The last component part of the artificial neuron is the sigmoidal transform which enables the neuron to produce a non-linear response. The circuit proposed in Figure 4.27 of §4.7.3 is implemented in `model` code Figure 6.41. The 80-bit E-sequence listing is omitted, it consists of connections for the load inputs of the shift register to either the power rail or to ground as appropriate. It is seen that a new 12-bit comparison is performed every 80 clock cycles, governed by the length of the E-sequence, which has the effect of reducing the frequency of the resulting output stochastic pulse signal. If this signal is fed into another neuron this problem should be ameliorated by the slicing action of the input weighting, but it will be most noticeable in the case of actually decoding the pulse stream.

Specifically testing the sigmoidal transform performance is difficult, however, the general operation can be determined by a similar exercise strategy to that of the input weight encoding. Three values corresponding to 0.2, 0.5 and 0.8 full scale are transformed using the circuit. A marked difference in the quantity of pulses should be seen between the three values transformed. The difficulty in producing more exact results is in performing the average of the output pulses by extraction from the output signal. Figure 6.42 displays the output waveform for this circuit. It can be seen that as the input values increase from $0.2 \rightarrow 0.5 \rightarrow 0.8$ at times 700000, 6900000 and 13002000 the density of the pulses in the output stream decreases. An error exists which has failed to be corrected in that the output of the transform should have been inverted. This has propagated throughout the whole artificial neuron design, fortunately a single inverter on the appropriate output pin cures this problem. This is the reason the output pulses become less dense rather than more dense.

6.3.9 The Whole Neuron

All the component parts required for an artificial neuron have now been designed and simulated. These circuits are interconnected appropriately to form the complete artificial neuron. In the process of compiling the whole design the power supply and ground are specified together with the input, output and bi-directional pad connections. This enables the remaining phases of the design stage (**gate**, **place** etc.) to be run for a unified ASIC design to be produced. The successful integration of all circuit elements enables a simulation of the artificial neuron to be performed.

Figure 6.43 displays the concise `model` code file for the complete neuron, for clarity all the pad interconnections off the ASIC are omitted. The benefit of the modular approach to design that Solo 1400 enables can be seen. Each sub-circuit has been designed and simulated before incorporation into a higher level component resulting in the complete neuron description in a limited number of lines of `model` code.

It was found that after the initial layout and routeing the physical die size was large and a core limited design had resulted ie. the size of the device is predominantly governed by the size of the chip array Figure 6.44. The smallest off the shelf package in which the die would fit was an 84-pin leadless chip carrier, LCC. A total of only 64 connections are necessary for a fully connected device as listed in Table 6.2 leaving 20 unused pins. As this

Signal	Quantity	Name	Type
Clock	1	Clk	Input
Read/Write	1	R/W	Input
Reset	1	Rst	Input
Weight Address	5	Addr(0:4)	Input
Weight Data	12	Init(0:11)	Bi-directional
Input Pulse Stream	16	In(0:15)	Input
Weight Up/Down Control	17	UD(0:16)	Input
Output Pulse Stream	1	Out	Output
Power Supply	5	Vdd(0:4)	Power
Ground	5	Gnd(0:4)	Power

Table 6.2: Necessary neuron connections

is a core limited device and rather than wasting the unused package connections additional output pads were added to the circuit to allow monitoring of internal areas of the device. In particular the output of the weight encoders were monitored `WghtOut(0:16)` and the result of the weighted input summation `SumOut`. This leaves just two unused pins. With the benefit of hindsight it would have been wise to have had a monitor on the output of the PRBS generator.

After recompilation of the circuit following the pad additions and the resulting progression through all the layout, routeing and packaging routines of Solo 1400 the pin connections resulting are listed in Table 6.3 and the pin layout is illustrated in Figure 6.45.

Although it is possible within the **mads** simulator to monitor internal nodes within the circuit the set of routines written to verify the performance of the neuron concentrated on the ability to only monitor the external connections since with a fabricated device probing internally would not be feasible. The set of tests created in the `wdl` file progress through the entire neuron exercising it in stages. Diagnostic style tests were included to verify operation of the internal circuits operation in case any problem occurred. The simulation consisted of several separate sections to test the address selection, the loading of input weight register values, the unloading of input weight register values, the ability of the weight encoders to convert the deterministic values into stochastic pulse streams, the summer operation and the sigmoidal transform. These tests are a reiteration of the tests conducted upon the sub-circuits but with the need to use the external chip connections and preceding circuits for driving the circuits under test. Having successfully verified the artificial neuron function the device is ready to be fabricated from the `.cif` file formed in the design and layout process. The fabrication has been conducted at a third party site through the EUROCHIP program.

Once the device has been fabricated it is necessary to test and verify the operation of the physical hardware. The hardware testing system employed is described in the following section §6.4. Following a description of the successful testing of an individual artificial neuron device a circuit is presented utilising six neurons operating to perform a simple standard task, the encoder/decoder problem.

6.4 Hardware Artificial Neuron Testing

To test the fabricated artificial neuron two hardware test configurations were considered.

1. The design and construction of a test board driven by a combination of signal generators and on board test circuits. Signals would be monitored and analysed via logic analysers and oscilloscopes.

2. The design and construction of a mounting circuit board with a cabling interface to a digital I/O card controlled from within a PC.

Each system does of course have its own advantages and disadvantages.

Considering first the construction of a test board driven by signal generators and monitored by logic analysers and oscilloscopes. The coordination of several pieces of external equipment to produce a unified test system becomes difficult. A total of 53 inputs are required for a neuron, though for some tests many are driven in parallel, the availability of equipment with the appropriate number of outputs becomes a problem. The ability

Pin	Name	Type	Pin	Name	Type
1	Gnd 0	Ground	43	UD 8	Input
2	OutWght 0	Output	44	UD 9	Input
3	OutWght 1	Output	45	UD 10	Input
4	OutWght 2	Output	46	UD 11	Input
5	OutWght 3	Output	47	UD 12	Input
6	OutWght 4	Output	48	UD 13	Input
7	OutWght 5	Output	49	UD 14	Input
8	OutWght 6	Output	50	UD 15	Input
9	OutWght 7	Output	51	UD 16	Input
10	OutWght 8	Output	52	Vdd 4	Power
11	Not Used	Not Used	53	Gnd 4	Ground
12	OutWght 9	Output	54	RW	Input
13	OutWght 10	Output	55	Vdd 3	Power
14	OutWght 11	Output	56	Gnd 3	Ground
15	Gnd 1	Ground	57	Clk	Control
16	Vdd 1	Power	58	Rst	Control
17	OutWght 12	Output	59	In 0	Input
18	OutWght 13	Output	60	In 1	Input
19	OutWght 14	Output	61	In 2	Input
20	OutWght 15	Output	62	In 3	Input
21	OutWght 16	Output	63	In 4	Input
22	SumOut	Output	64	In 5	Input
23	Out	Output	65	In 6	Input
24	Init 9	Bi-dir	66	In 7	Input
25	Init 10	Bi-dir	67	In 8	Input
26	Init 19	Bi-dir	68	In 9	Input
27	Addr 0	Control	69	In 10	Input
28	Addr 1	Control	70	In 11	Input
29	Addr 2	Control	71	In 12	Input
30	Addr 3	Control	72	In 13	Input
31	Addr 4	Control	73	In 14	Input
32	UD 0	Input	74	Init 0	Bi-dir
33	Not Used	Not Used	75	Init 1	Bi-dir
34	UD 1	Input	76	Init 1	Bi-dir
35	UD 2	Input	77	Init 2	Bi-dir
36	UD 3	Input	78	Init 3	Bi-dir
37	UD 4	Input	79	Init 4	Bi-dir
38	UD 5	Input	80	Init 5	Bi-dir
39	UD 6	Input	81	Init 6	Bi-dir
40	Vdd 2	Power	82	Init 7	Bi-dir
41	Gnd 2	Ground	83	Init 8	Bi-dir
42	UD 7	Input	84	Vdd 0	Power

Table 6.3: Artificial neuron chip pin connections

to control the time between operations and signal changes is a definite advantage as are the measurement capabilities provided by a logic analyser. If several pieces of equipment are used for driving the device then synchronisation may become a problem. Sampling of output lines with the resulting pulse counting and averaging will not be straightforward with this system.

If the second system is adopted a basic breakout of the ASIC pins to connectors is required which will link to a PC driven digital I/O card under software control. A highly versatile system will result for the controlling, driving and reading from the communication lines. The timing information between signals will be limited by the timing capabilities written into the software. The signal level monitoring, accumulation of output pulses and processing will be straightforward as this can all be handled by the software. It is still feasible to use an oscilloscope and logic analyser as external pieces of test equipment for verifying signal performance if required. The basic trade-off between the two approaches is hardware complexity vs software complexity.

It was decided to adopt the second system of testing due to the expected relative short lead time for fabrication of the board and generation of the test software. The simple hardware test layout is illustrated in Figure 6.47 where two FPC-024 digital I/O cards were installed in a PC allowing a maximum of 96 lines to be controlled in four groups of three sets of eight lines. Appendix D lists the 72 interconnections necessary between the I/O cards and the neuron chip.

To control, read from and write to these lines through the digital I/O cards software written using C++ was produced. C++ was chosen since it would allow the development of a simple class for the digital I/O cards.

The testing software written can be broken down into three areas

1. The FPC-024 class for driving the digital I/O cards.
2. A set of library routines for controlling specific lines eg. CLK, RST, as well as more complex routines for loading and unloading weight values for a given input signal.
3. The test routines written to exercise the neuron which are built from the component routines of (1) and (2).

The test routines will now be individually described and discussed.

`testWghts()` To be able to successfully use the neuron the input weight register must be able to be written to and read from. With all the data inputs set `DATA_HIGH` and the up/down lines set to `COUNT_UP` each of the 17 weight register is loaded with a preset value in turn. The weight registers are then immediately unloaded in turn. The result is that the unloaded value is 16 more than the value originally loaded since each register will have been clocked 16 times between loading and unloading. The test also confirms the operation of the address selector, multiplexors and demultiplexors through the bi-directional sections of the chip.

`testCountUp()` Each weight register is tested to verify that the counter will count up a specified number. With the data value set `DATA_HIGH` and the count direction set at `COUNT_UP` the weight register is loaded with a mid-range positive value and then clocked a known number of cycles before reading the value back out. The read out value from the weight register should be the number of clock cycles in excess of the value originally loaded in. This test is repeated for a mid-range negative value. All 17 weight registers are tested in this manner.

`testCountDown()` This is a companion test to `testCountUp()` in that the same procedure is followed to test the 17 weight registers except that the count direction is set to `COUNT_DOWN` and the value read back in should be the appropriate number of clock cycles less than the value originally loaded in.

`testZeroCross()` The zero crossing is tested for each of the weight registers. A value less than zero is loaded with the count direction set to `COUNT_UP` and the counter clocked through zero for a known number of cycles and the correct positive value is read back out. The chip is reset and loaded with a value just greater than zero with the count direction set to `COUNT_DOWN` the counter is clocked back through zero for another known number of cycles and the correct negative value read back out.

`testDirChange()` Taking each weight register in turn the register is loaded with a mid-range positive value. The register is set to `COUNT_UP` and the register is clocked for a known number of pulses. The direction of the count is reversed to `COUNT_DOWN` and the register clocked another known number of pulses. Finally the count direction is reset back to `COUNT_UP` and the counter clocked for a final number of known cycles. At each change of count direction and at the end of the test the value of the weight register is read out and confirmed to be correct. The aim of this test is to verify that as the direction of count is changed while the counter is in use the counter correctly changes direction without any loss or gain in its value. The test strategy is repeated for both for each weight register and in the negative half of the counter range.

`testMaxLimit()` The aim of this test is to confirm that each weight register will count up to its maximum value of 2047 and then stop until the direction of the count reverses to `COUNT_DOWN` at which point the register should move down.

This test initially failed in that the counters correctly increment to their maximum limit and stop but on reversal of the count direction they clock over from the maximum value to their minimum value at which point the counter is being driven to `COUNT_DOWN` and so holds its value at the minimum value. This led to a rethink of the clocking and driving strategy such that the time the up/down line changes occurs when the clock is at `CLOCK_HIGH` rather than `CLOCK_LOW` as previously. The weight counter then correctly stopped at the maximum value and counted down when the direction of the up/down signal reversed.

`testMinLimit()` This is a companion test to `testMaxLimit()` in that a similar procedure is used to test the limit stop at the lower end of the count -2048. The weight register under test is initially loaded with a value just greater than the minimum limit and set to `COUNT_DOWN`. Before the timing changes for the direction of the up/down signal had been corrected the counter would stop at -2048 until the direction of count reversed at which time it would clock over to 2047 where it would be attempting to `COUNT_UP` and then the counter would again halt.

`testWghtEncode()` Three values are loaded in turn into each weight register -1024, 0, 1024. For each of these values the circuit is clocked sufficient times to produce a `RUN_LENGTH` long output sequence. In effect the number of clock cycles in 12 $\times$ `RUN_LENGTH`. The output of the pulse coded value from the weight encoder circuit `WghtOut(*)` is sampled every 12 clock cycles after each comparison has been performed. The accumulated output pulses divided by the `RUN_LENGTH` is a measure of the encoded value given by eq.(4.7). For the three values above the results will be approximately 0.25, 0.5, 0.75 respectively. The accuracy of this result will depend upon the actual `RUN_LENGTH`. The greater the value of `RUN_LENGTH` the better the estimate to the desired value.

During the testing the input lines are all set to `DATA_HIGH`. The up/down control lines are toggled after every clock pulse so that the weight register counts up by one and then counts down by one thus maintaining a constant value for encoding.

`testPulseDivider()` To verify that each of the 17 signals input to the pulse divider circuit preceding the summer is weighted by $\frac{1}{17}$ the corresponding input weight register is loaded with its maximum value, the input is set to `DATA_HIGH` and the direction of count set to `COUNT_UP`. This will cause a permanent high signal to be the resulting weighted input. All the other inputs are set to `DATA_LOW`, their count direction set to `COUNT_DOWN` and their weight register loaded with the minimum value. This causes a permanent low signal to be output by the resulting weighted input.

Only one input to the pulse divider circuit will be high and the pulse divider output for this signal will be value of the weighting applied to it, $\frac{1}{17}$. This is monitored at `SumOut` the output of the summer which will not be affected by the other inputs as they are all low. By cycling through which of the 17 weighted inputs is high the 17 pulse divider signals can be tested.

`testSummer()` By a simple extension of the ideas of the previous test `testPulseDivider()` of setting an input to the pulse divider permanently high, by setting several permanently high fixed addition in steps of $\frac{1}{17}$ can take place through the summer. Thus to test the summer the number of signals permanently high is ramped up and the series $0, \frac{1}{17}, \frac{2}{17}, \ldots \frac{17}{17}$ can be measured at the pin `SumOut`.

`testSigmoid()` This final neuron test routine builds upon the previous routine `testSummer()` in that the actual neuron output `Out` is monitored as the number of inputs set high is ramped up. The value of `Out` has to be sampled every 80 clock pulses since the speed of update is governed by the E-sequence length in the Gaussian random number generator which is 80-bits long. The value read out will be inverted ie. `1 - actual value` but this can be easily corrected by addition of an inverter in the practical circuit usage of this device.

6.5 A 4–2–4 Encoder/Decoder Implementation

To be able to demonstrate the capabilities of the artificial neuron device operating in a coherent manner a proposal to design a dedicated hardware network utilising six of the fabricated neurons was put forward and implemented. This proposal was set aside at a late stage due to unsurmountable communication problems with each individual neuron. A second, successful, approach was attempted by writing appropriate driver software to simulate the operation of a network of six neurons by multiplexing the operation through a single neuron on the test board of §6.4. A short description of the original proposal will be given due to to the effort expended upon it. This section will then move onto the successful multiplexed system implementation, a description of the weights used to perform the task and the results of operating the network.

6.5.1 System Implementation: 1st Proposal

The first proposal was to use the experience gained in the single test board to design and build a network of six neuron boards mounted on a backplane motherboard, Figure 6.48. Control of the system would be effected through the two FPC-024 digital I/O cards as per the individual neuron test board of §6.4. The addressing space would need to be extended to allow each neuron board to be addressed independently. Monitoring of individual weight encoding procedures would no longer be possible without a significant increase in wiring complexity or switching circuitry. Each neuron's operation will have been verified initially using the neuron test board. Each neuron's output was however directly monitored. Appendix E contains the digital I/O card connections and the circuit diagrams for the dedicated hardware.

After fabrication of the six neuron boards, backplane and writing of the main driver software, communication between the ASIC socket and the ASIC was found to be intermittent, irregular and lacking continuity. Several sockets from different suppliers were tested but none with satisfactory results. This problem had been encountered with the test board but had been accounted for by the use of a cheap, poorly specified socket. A special purpose ZIF (Zero Insertion Force) test connector had been used for the test board to overcome this problem. The cost and size of ZIF sockets are prohibitive for their use in situations other than as a reusable ASIC chip mount. It was this problem of poor continuity which led to the design ultimately being set aside.

6.5.2 System Implementation: 2nd Proposal

The second, less visually effective, proposal to demonstrate coherent network operation was to re-utilise the test board. A network of neurons can be simulated by time-multiplexing the operation through a single device. Reference to Figure 6.49 a 4–2–4 Encoder/Decoder feedforward configuration will aid in understanding the following description. Initial input sequences for the network are generated and held in arrays on the host PC. Since the four input neurons act as purely distribution points for information Neuron 1, in the hidden layer, is the first to be driven. The weights, scaled appropriately, for the neuron are initialised to those necessary for such a hidden neuron and the four input pulse sequences fed into the neuron. As each input pulse combination is processed the single output pulse is stored in an array on the host. Once the input sequence has been exhausted the single neuron is loaded with the weights appropriate for Neuron 2 and the four input sequences passed through the neuron with the storage of the single output pulse stream in a new array on the host.

To process the output layer neurons, 3–6, the process of running pulses through one neuron at a time and storage of the output pulse stream is repeated. On these occasions though the pulse sequence is to be input are taken from the two output sequence arrays for the hidden layer neurons. Decoding of the output pulse sequence can be undertaken to verify that they are the correct value.

If longer input pulse sequences are required ie. the network is to be run over a longer time frame, a fresh set of four input streams can be generated and the multiplexing process can be continued as often as desired. The output value of the network would then need to be taken over the effective full output sequence length or a software implementation of one of the output processes of §4.8 used.

By the use of the multiplexing technique it is feasible to describe and run a feedforward network of arbitrary size for network evaluation purposes. It would not be possible to adjust weights on-line, each neuron's weight would need to be pre-determined.

6.5.3 Weight Determination

For the demonstration network of the 4–2–4 encoder/decoder network no on-line adaption was to be performed. The weight values for each neuron were to be determined in advance and loaded in as required, (all at once in the first proposal, one neuron at a time in the

Neuron	Weight	Matlab Value	Scaled Values
1	Bias	-0.1221	-81
1	1	0.7312	486
1	2	2.3173	1452
1	3	-2.4702	-1644
1	4	-1.4920	-993
2	Bias	-0.6493	-432
2	1	-1.3388	-891
2	2	2.1569	1436
2	3	-3.0768	-2048
2	4	1.6528	1100
3	Bias	-2.7767	-2008
3	1	2.8239	2042
3	2	-2.8204	-2039
4	Bias	-2.7953	-2021
4	1	2.8107	2032
4	2	2.7569	1993
5	Bias	-2.7706	-2003
5	1	-2.8014	-2025
5	2	-2.7512	-1989
6	Bias	-2.7854	-2014
6	1	-2.8272	-2044
6	2	2.8327	2048

Table 6.4: Possible weight values to be loaded into each neuron as determined by the use of a network trained using Matlab. *The Scaled Values are those which are to be loaded into the hardware neuron.*

second). The values the weights should take could be determined by the use of commonly available software using the backpropagation learning algorithm for this form of network. Using the Neural Network Toolbox in Matlab a set of possible weights could be determined as shown in Table 6.4. Problems will exist with these learned values since although they operate with a small error in the simulation they do not account for the specific shape of the sigmoid in the hardware, neither do they account for the reduced output range of the hardware neurons caused by only a proportion of the inputs being used.

It is known for this problem of encoding and decoding that the hidden layer neuron weights are such that the hidden layer neurons produce a binary representation of the input line which is high. The output layer neuron weights are such that the hidden layer binary representation is decoded back to a single line being high. An appropriate set of weights for the neuron can thus be configured as shown in Table 6.5. These values should overcome the limitations of the sigmoid not producing an adequate squashing function and the limited dynamic range of the output.

Neuron	Weight	Weight Value
1	Bias	0
1	1	-2040
1	2	2040
1	3	-2040
1	4	2040
2	Bias	0
2	1	-2040
2	2	-2040
2	3	2040
2	4	2040
3	Bias	-512
3	1	-2040
3	2	-2040
4	Bias	-512
4	1	2040
4	2	-2040
5	Bias	-512
5	1	-2040
5	2	2040
6	Bias	-512
6	1	2040
6	2	2040

Table 6.5: Weight values for 4–2–4 hardware encoder/decoder. *These values are determined by a combination of inspection of the problem and the solution of the equations which describe the system.*

6.5.4 Results of System Operation

After the transition from a system of six neurons all operating coherently to a single neuron simulating the operation of the six by multiplexing its operation it was possible to demonstrate the system operation. The above second proposal of time multiplexing process was successfully implemented in software and the single neuron driven in order to demonstrate the 4–2–4 encoder/decoder. The drawback of this approach is that the network took six times as long to operate and the benefit of parallel operation is obviously lost.

The system was first driven with the 'learned' weight values from the Matlab simulation. Table 6.6 displays the results of this network when run. It can be seen that the average output values of the neurons are close to 0.5 equivalent to zero when converted from the SLB representation to a real value. Applying the decoding transform of eq.(4.9) it can be seen that the hidden layer, neurons 1 and 2, does indeed have a binary representation of the input lines being high. However, this does not continue through to the appropriate line being high for the output layer, neurons 3, 4, 5 and 6.

With the new set of weights, illustrated in Table 6.5, the neuron outputs are as shown in Table 6.7. Again a binary coding of input values is evident in Neurons 1 and 2 of the hidden layer. This time they result in the appropriate output layer neuron firing and being high, neurons 3, 4, 5 and 6.

On re-inspecting the two sets of weight values in Table 6.4 and Table 6.5 it can be seen that the form of the weight values are of approximately the same configuration with respect to sign and magnitude. The determined values of Table 6.5 simply drive the neurons harder to the limits of the output to overcome the poor sigmoid.

A drawback in the N input adder was observed that had not been previously considered. When less than a full number of inputs are used, the unused inputs being set to a value of zero, the range of output values from the adder will restricted to the proportion of inputs actually used due to the constant $\frac{1}{N}$ scaling. Thus, if only n of the maximum N inputs are used the swing in output value of the adder will be $\frac{n}{N}$.

6.6 Summary

In this chapter we have used the ideas and techniques of the previous two chapters §4 and §5 to present a novel design of an artificial neuron operating by the use of stochastic pulse rate encoded signals. The neuron design has been implemented in CMOS VLSI using the Solo 1400 design package in 1.5μm technology. The design uses approximately 5500 gates and 27000 stages which covers an active chip area of $9.59 \times 8.13 = 77.98$sq.mm

This chapter began with a specification for a 16 input device operating using SLB signals and a block diagram of the artificial neuron circuit to be designed. An evaluation of the design system options available was made which resulted in the selection of the

Input Configuration	Neuron	Output Value	Converted Output Value
1, 0, 0, 0	1	0.475356	-1
	2	0.537719	1
	3	0.528606	1
	4	0.473375	0
	5	0.471869	0
	6	0.468931	0
0, 1, 0, 0	1	0.444369	-1
	2	0.473919	-1
	3	0.498350	0
	4	0.501975	0
	5	0.443256	0
	6	0.438931	0
0, 0, 1, 0	1	0.534744	1
	2	0.512819	1
	3	0.442706	0
	4	0.440363	0
	5	0.499894	0
	6	0.499156	0
0, 0, 0, 1	1	0.515656	1
	2	0.483288	-1
	3	0.440612	0
	4	0.441631	0
	5	0.499956	0
	6	0.497706	0

Table 6.6: Neuron output values for the four input schemes possible with a 4–2–4 encoder/decoder, trained weights. *Hidden layer neuron output values are converted on the basis of the sigmoid, while the output layer values have been thresholded at* $T = 0$. *NB. The neuron outputs are SLB representation therefore an output of 0.5 translates to an actual value of 0.*

Input Configuration	Neuron	Output Value	Converted Output Value
1, 0, 0, 0	1	0.555506	1
	2	0.556350	1
	3	0.552144	1
	4	0.493888	0
	5	0.491812	0
	6	0.432456	0
0, 1, 0, 0	1	0.440656	-1
	2	0.556613	1
	3	0.494569	0
	4	0.554513	1
	5	0.434725	0
	6	0.491325	0
0, 0, 1, 0	1	0.556394	1
	2	0.442463	-1
	3	0.493244	0
	4	0.436450	0
	5	0.551087	1
	6	0.491881	0
0, 0, 0, 1	1	0.437956	-1
	2	0.442794	-1
	3	0.435781	0
	4	0.497350	0
	5	0.494100	0
	6	0.551037	1

Table 6.7: Neuron output values for the four input schemes possible with a 4–2–4 encoder/decoder, calculated weights. *Hidden layer neuron output values are converted on the basis of the sigmoid, while the output layer values have been thresholded at $T = 0$. NB. The neuron outputs are SLB representation therefore an output of 0.5 translates to an actual value of 0.*

Solo 1400 design package §6.2. Following a description of Solo 1400's main tools to be used in the design process a detailed description of the neuron sub-circuits is made consisting of either schematic diagrams or HDL descriptions of the circuits §6.3. Simulation test files are presented with their resulting output which demonstrate the correct operation of the sub-circuits. The sub-circuits are combined to form a complete artificial neuron which has subsequently been fabricated.

In section §6.4 the testing system for the fabricated device is outlined together with a description of the software test routines used to exercise the device. Due to the nature of the testing system the operation of the device is limited to basically a yes no response. The artificial neuron device operates as desired producing a weighted sum of 16 inputs using stochastic pulse rate encoded processing. However the non-linear sigmoidal transform is limited use due to its poor performance.

Section 6.5 describes how the hardware neurons which have developed throughout this chapter have been configured into a small example network to perform the 4–2–4 encoder/decoder problem. Two systems were attempted, the first unsuccessful system used six devices operating in a parallel, the second successful approach used a single neuron through which all the necessary signalling was multiplexed. The first system proved unsuccessful because clear and consistent connectivity could not be achieved to all the designed neuron boards. The bad connectivity has been attributed to a poor design in the ASIC packaging and associated connector. The second system reused the test board in the previous chapter but with new driver software.

Given an appropriate set of neuron weight values it was demonstrated that the network of six neurons could perform the 4–2–4 problem. A set of weights obtained by training a model of the system in software using backpropagation were found not to be adequate since they did not drive the neurons sufficiently hard. The model of the sigmoid would need to be more precise for accurate off-line training to be performed. Using a semi-heuristic technique to find an alternative set of weights which drove all the neurons either fully-on or fully-off the network was able to more clearly demonstrate the performance of the task.

This system implementation highlights several areas of work which may be developed further: the formation of an N input adder which does not suffer from the scaling difficulties, the formation of an improved sigmoid transform and the development of an accurate functional model of neuron to enable software simulation of its performance and off-line training to be performed if desired.

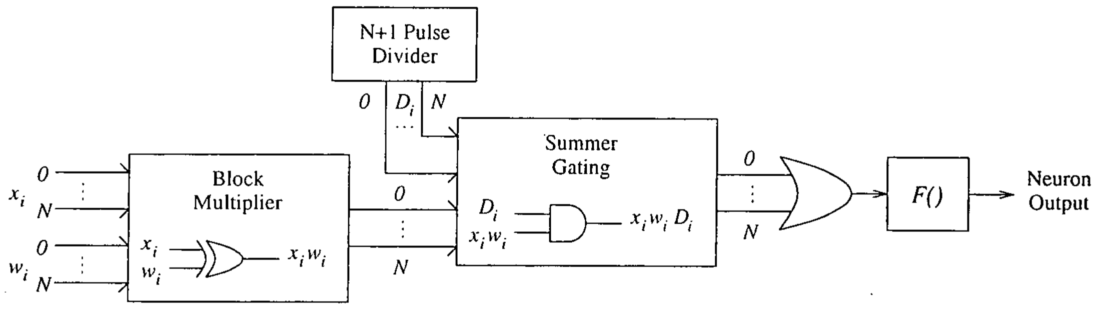


Figure 6.1: Basic architecture for a stochastic pulse neuron. *The neuron produces a function of a weighted sum of inputs. Signals are of the single line bipolar form.*

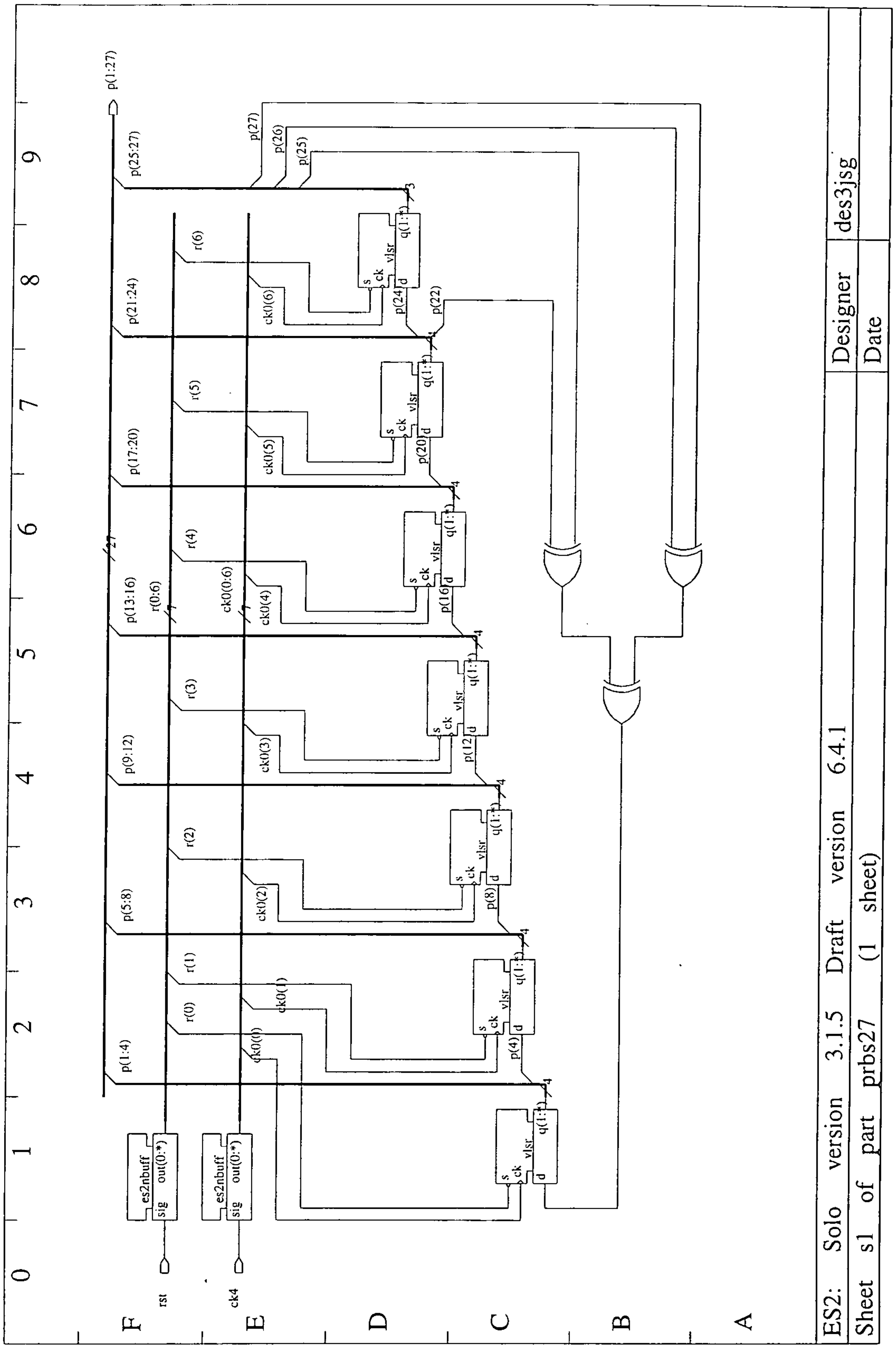


Figure 6.2: 27-bit PRBS generator schematic. *The basic shift register is composed of vlsr building blocks with an* **XOR** *feedback circuit.*

```
{ ##########
{ part:           vlsr
{
{ description:  Variable length schift register.
{                   Adapted from ES2 example, each stage
{                   outputs its value. An asynchronous set is used to
{                   set the device to all 1's.
{ ##########
Part vlsr (n) [ck,d,s] → q(1:n)

    Integer i

    Signal qi(1:n+1)

    If n = 0 Then
        d → q(1)
    Else
        d → qi(1)
        For i = 1 : n Cycle
            bdffs [ck,qi(i),s] → qi(i+1), --
            qi(i+1) → q(i)
        Repeat
    Endif
End                                                    { end of Part vlsr declaration
```

Figure 6.3: `model` code for variable length shift register. *Specifying the variable n determines the length of the shift register. Note the* **For - Repeat** *loop construct simplifying the design specification.*

```
{ ##########
{ part:                    prbs27to38
{
{ description:          Forms 38 sequences from 27 bit PRBS generator.
{ ##########
Part prbs27to38 [in(1:27)] → prbsout(1:38)

  Signal inbuf(1:27),
         link(0:82)

  arraybuffer (27) [in(1:27)] → inbuf(1:27)

{ prbsout(1)
  xor [inbuf(21),inbuf(24)] → link(0)
  xor [inbuf(25),inbuf(26)] → link(1)
  xor [link(0),link(1)] → prbsout(1)

{ prbsout(2)
  xor [inbuf(8),inbuf(10)] → link(2)
  xor [inbuf(19),inbuf(21)] → link(3)
  xor [link(2),link(3)] → prbsout(2)

{ prbsout(3)
  xor [inbuf(11),inbuf(18)] → link(4)
  xor [link(4),inbuf(23)] → prbsout(3)
```

Figure 6.4: Sample `model` code for 38 taps offs from 27-bit PRBS. *No input variables to configure this stage were possible, each gate has to be specified and connected explicitly.*

```
// #########
// file:          prbs27.wdl
//
// description:          Test and exercise 27 bit Pseudo Random Binary
//                  Sequence Generator.
// #########

// #########
// function:      clkPulse()
//
// description: toggle a signal line twice, normally the clock
// #########
void
clkPulse(signal clock)
{
         Toggle(clock);
         Simulate;
         Toggle(clock);
         Simulate;
}  // end of function clkPulse()

// #########
// main function to exercise the 27-bit PRBS
// #########
main()
{
// control lines
         Input ck4;
         Input rst;

// data lines
         Output p(27:1);

         Set_Cycle(1000);

// initialise prbs27
         ck4 = 1;
         rst = 1;
         clkPulse(ck4);
         rst = 0;
         clkPulse(ck4);
         rst = 1;

// run prbs27 for 50 clock cycles
         for (i = 0; i < 50; i++) {
                  clkPulse(ck4);
         }  // for i

// reset prbs27 and run again
         rst = 0;
         clkPulse(ck4);
         rst = 1;

// run prbs27 for 50 clock cycles
         for (i = 0; i < 50; i++) {
                  clkPulse(ck4);
         }  // for i

}  // end of main()

// #########
```

Figure 6.5: `wdl` code for exercising 27-bit PRBS. *After initialising all the input lines the PRBS is clocked for 50 cycles before being reset and clocked for another 50 cycles. Note the use of procedures in the code.*

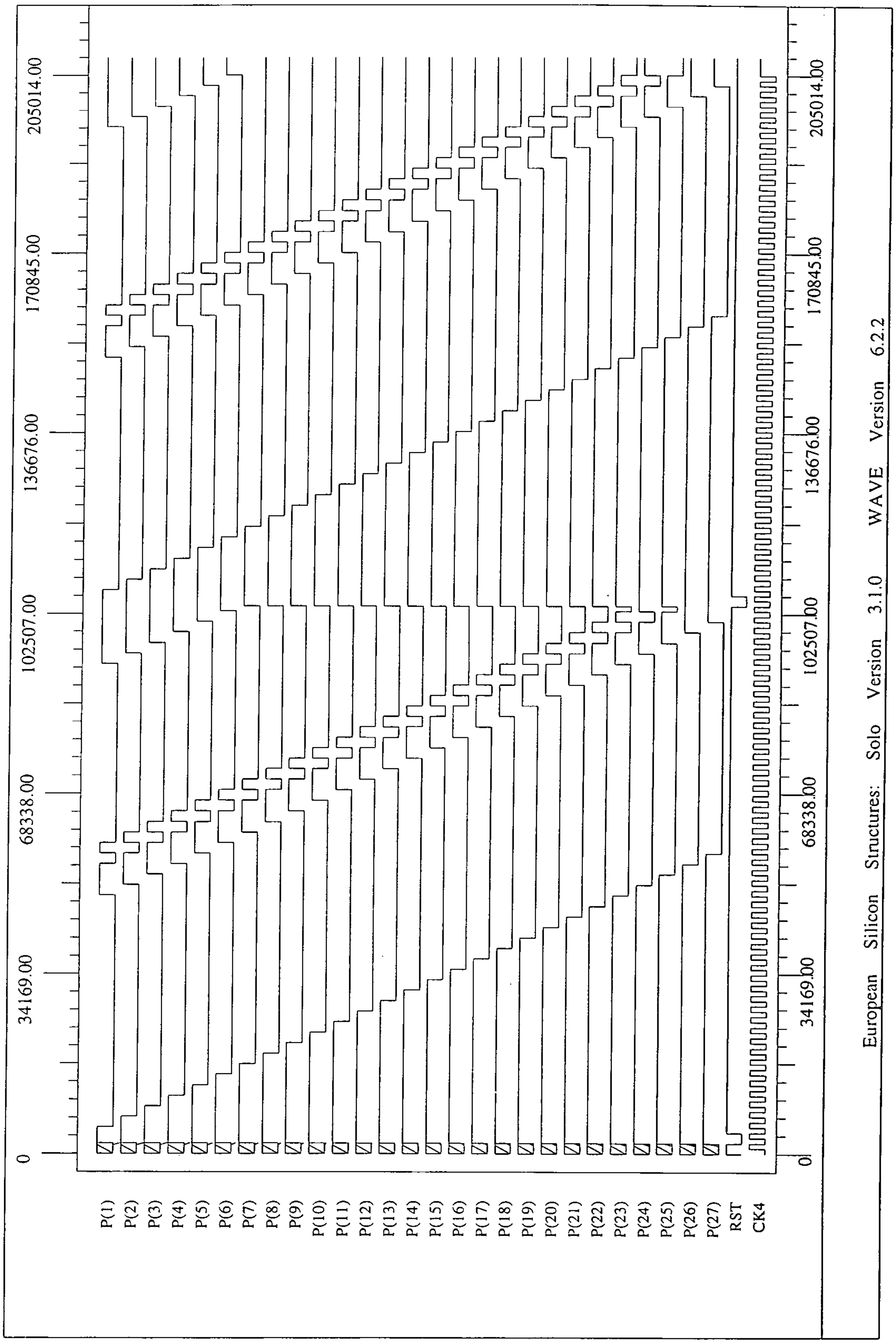


Figure 6.6: **wave** output plot for 27-bit PRBS generator. *After the initialisation phase the PRBS has been clocked for 50 cycles, the pulse train can be seen to ripple through the shift register. Following the reset of the PRBS the same sequence of pulses is repeated.*

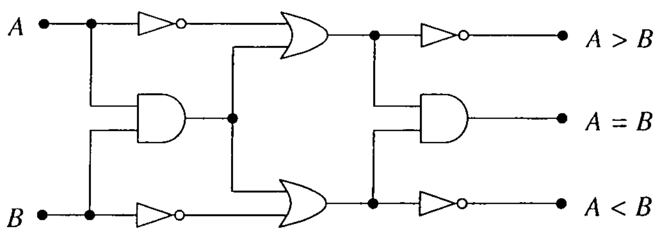


Figure 6.7: One-bit comparator. *Simple combinational logic circuit for comparing two inputs.*

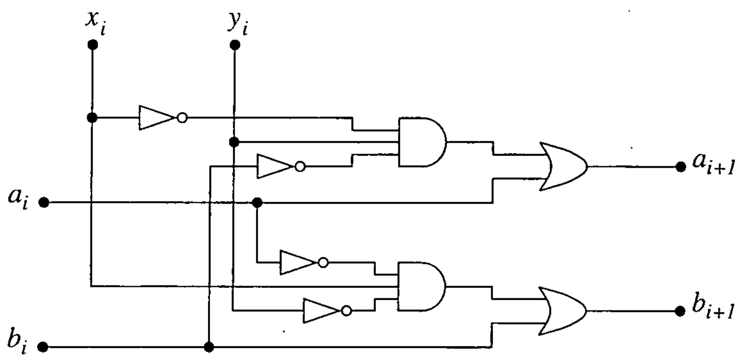


Figure 6.8: Iterative comparator cell. *Combinational logic building block which will use current line values together with the previous result to generate a comparison output.*

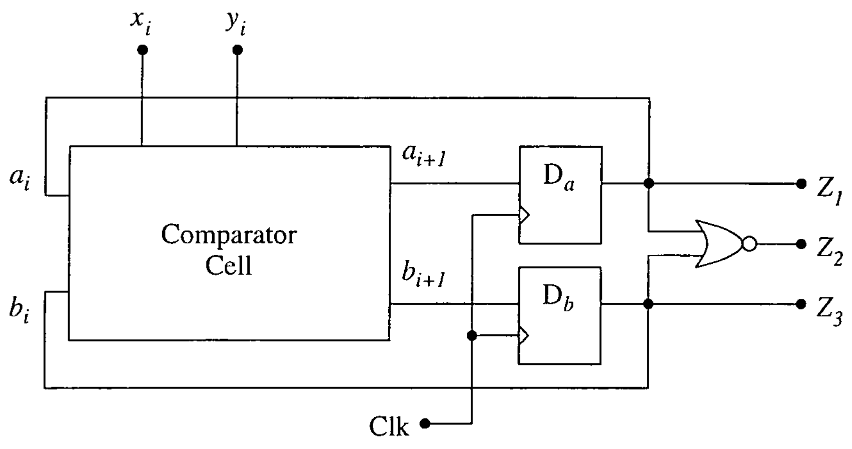


Figure 6.9: Iterative comparator. *Sequential logic circuit utilising the comparator cell of Figure 6.8 and D-type flip-flops for storing the results of the comparison.*

```
{ #########
{ part:            comp_cell
{
{ description:  Combinational logic for an iterative comparator
{ #########
Part comp_cell [nota,notb,x,y] → aout,bout

    Signal notx,
           noty,
           bxyout,
           axyout

    not [x] → notx
    not [y] → noty

    nand [notb,notx,y] → bxyout : nand3_bxyout
    nand [nota,x,noty] → axyout : nand3_axyout

    nand [bxyout,nota] → aout : nand2_aout
    nand [axyout,notb] → bout : nand2_bout

End                                         { end of Part comp_cell declaration
```

Figure 6.10: model code for iterative comparator building block. *Implementation of the comparator cell of, Figure 6.8. Note that the circuit has been organised to use simply* **NAND** *gates.*

```
{ #########
{ part:            comp_iter
{
{ description:   An Iterative Comparator
{                  Two control lines clk & rst
{                  Output of comparison changes on rising edge of clk
{                  Output reset to r = 0 & t = 0 when rst held low
{                  Two data inputs x & y
{                  Three outputs r & t which are defined as follows
{                    r == x < y        t == x > y
{ #########
Part comp_iter [clk,x,y,rst] → r,t

    Signal aout, bout,
           notr, nott

    comp_cell [notr,nott,x,y] → aout,bout

    dff rn [clk,aout,rst] → r,notr
    dff rn [clk,bout,rst] → t,nott

End                                         { end of Part comp_iter declarartion
```

Figure 6.11: model code for complete iterative comparator. *Note how the modular design process enables the previously produced module, Figure 6.10, to be included and connected up to the additional flip-flops.*

```
// ##########
// file:          comp_test.wdl
//
// description:          Test and exercise iterative comparator circuit.
// ##########

// main function to test the comparator for 3 cases,
// only 4-bit numbers used but can be extended to n-bit numbers.
// ##########
main()
{
  Input x; Input y;
  Input clk; Input rst;
  Output r; Input t;

  Set_Cycle(50);

  x = 0; y = 0;
  clk = 0; rst = 1;

  Toggle(clk);
  Simulate;
  clkPulse(clk); { defined in a previous wdl file
  rst = 0;
  clkPulse(clk);
  rst = 1;

//  x < y          x: 1010  y: 1100
   x = 1; y = 1; clkPulse(clk);
   x = 0; y = 1; clkPulse(clk);
   x = 1; y = 0; clkPulse(clk);
   x = 0; y = 0; clkPulse(clk);

// reset comparator
  rst = 0;
  clkPulse(clk);
  rst = 1;

// x > y          x: 1101  y:1011
  x = 1; y = 1; clkPulse(clk);
  x = 1; y = 0; clkPulse(clk);
  x = 0; y = 1; clkPulse(clk);
  x = 1; y = 1; clkPulse(clk);

//  reset comparator
  rst = 0;
  clkPulse(clk);
  rst = 1;

// x == y          x: 0101  y: 0101
  x = 0; y = 0; clkPulse(clk);
  x = 1; y = 1; clkPulse(clk);
  x = 0; y = 0; clkPulse(clk);
  x = 1; y = 1; clkPulse(clk);

}  // end of main()
```

Figure 6.12: wdl code for testing iterative comparator. *Three 4-bit tests are run, one for each of the possible input cases.*

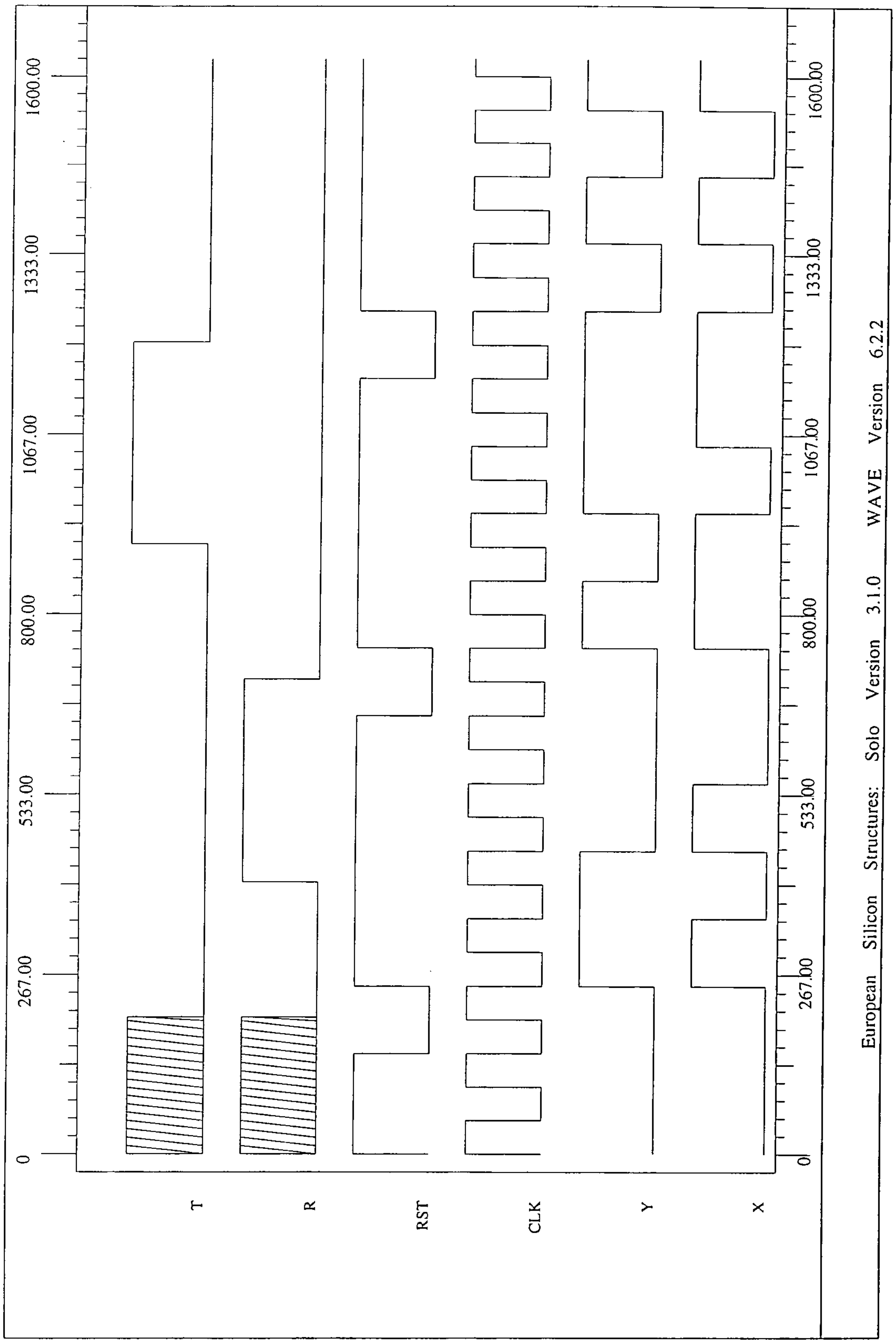


Figure 6.13: **wave** output plot for iterative comparator. *The three separate input cases for X and Y can be seen applied, one after each reset of the comparator. The appropriate result is visible as an output high on R or T.*

```
{ ##########
{ part:             count12a
{
{ description:  4 bit counter which counts up to 12 and resets to 0
{ ##########
Part count12a [clk, rst] → cnt12a

  Signal sumlsbs, count(0:3)
  Signal rstcnt, rstcs

  Probe sumlsbs
  Probe rstcnt
  Probe rstcs
  Probe count(0:3)

  and [count(0:1)] → sumlsbs
  ornand (1,2) [count(3),count(2),sumlsbs] → rstcnt
  and [rst,rstcnt] → rstcs
  es2ctr (4) [clk,Gnd,Gnd,Gnd,Gnd,Gnd,Vdd,rstcs] → count(0:3)
  bdff [clk,rstcnt] → --, cnt12a

End                                          { end of Part count12a declaration
```

Figure 6.14: `model` code for count12. *A 4-bit counter, es2ctr, is used which has external combinational logic to reset itself and generate an output when the circuit reaches 12.*

```
{ ##########
{ part:            count80
{
{ description:  7 bit counter which counts up to 80 and resets to 0
{ ##########
Part count80 [clk, rst] → cnt80

  Signal sumlsbs, count(0:6)
  Signal rstcnt, rstcs

  and [count(0:3)] → sumlsbs
  ornand (1,3) [count(6),count(5),count(4),sumlsbs] → rstcnt
  and [rst,rstcnt] → rstcs
  es2ctr (7) [clk,Gnd,Gnd,Gnd,Gnd,Gnd,Gnd,Gnd,Gnd,Vdd,rstcs] → count(0:6)
  bdff [clk,rstcnt] → --,cnt80

End                                                    { end of part count80
```

Figure 6.15: `model` code for count80. *This is a variant of Figure 6.14. A 7-bit counter is used which has external combinational logic to reset itself and generate an output when the circuit reaches 80.*

```
// #########
// file:          count12atst.wdl
//
// description: test an exercise up counter to 12
// ########

main()
{
        Input clk;
        Input rst;

        Output cnt12a;

        Set_Cycle(1000); { described in earlier wdl file

        clk = 1;
        rst = 0;
        clkPulse(clk);
        rst = 1;

        for (i = 0; i < 60; i++) {
                if (!(i % 33)) {
                        rst = 0;
                }  // if i
                if (!((i - 1) % 33)) {
                        rst = 1;
                }  // if i
                clkPulse(clk);
        }  // for i

}  // end main()
```

Figure 6.16: wdl code for testing the count12. *After reseting the counter, it is clocked to verify it counts and reinitialises itself before undergoing an external reset and continued clocking.*

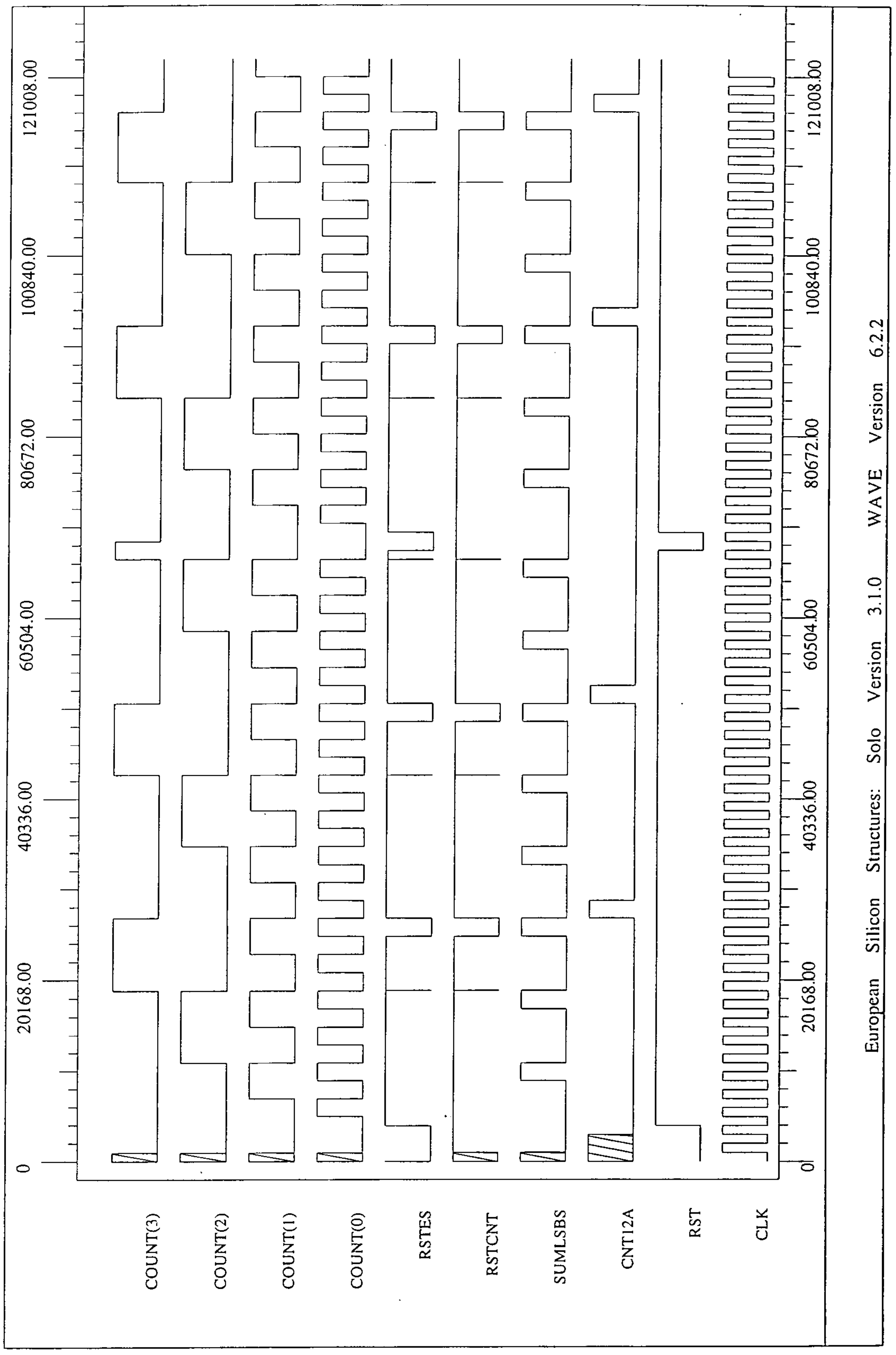


Figure 6.17: **wave** output plot for count12a testing. *The individual bits of the counter can be seen to count up, while the output, CNT12A, only goes high after 12 cycles except when the circuit is reset externally on the RST line.*

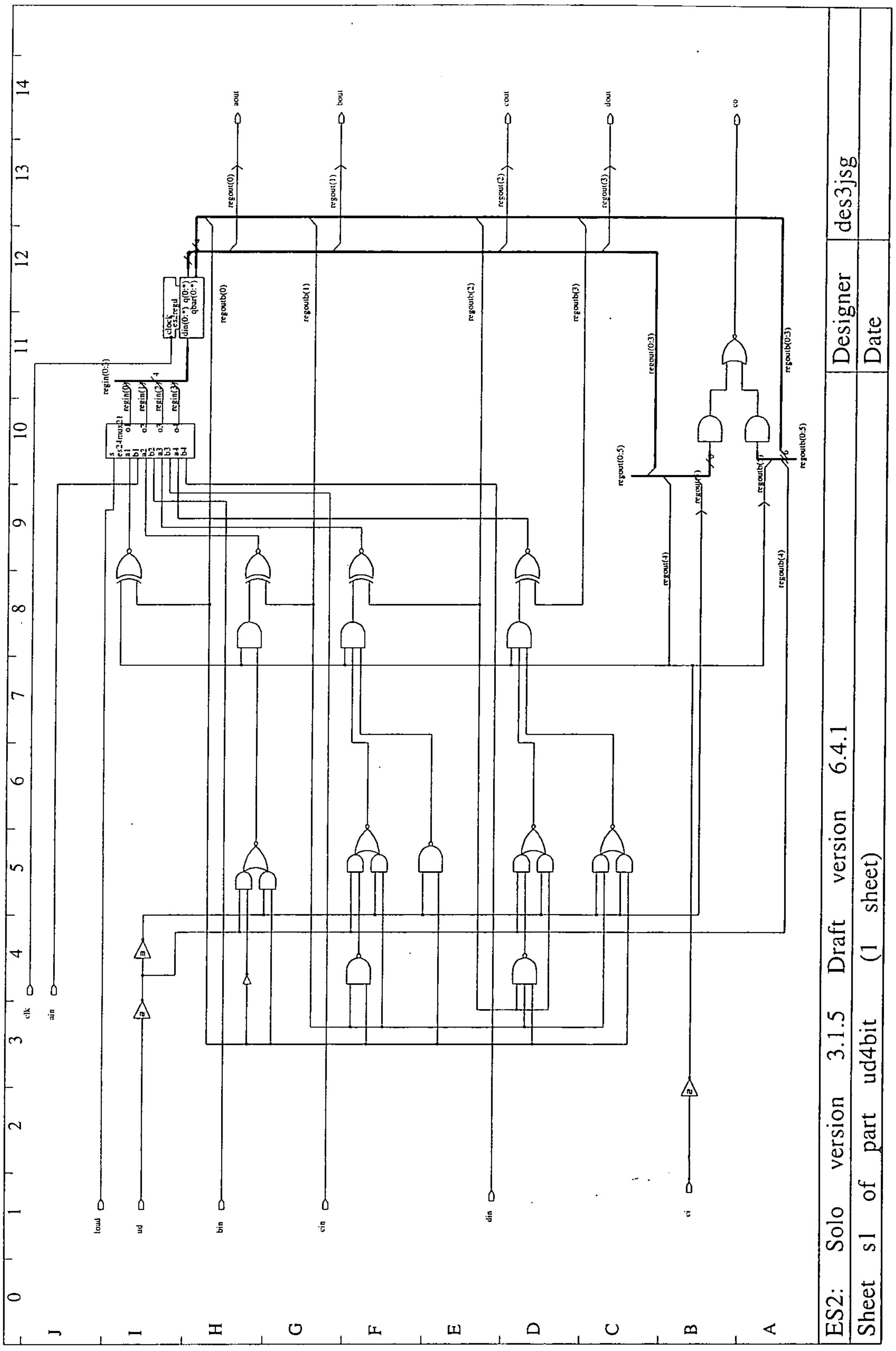


Figure 6.18: 4-bit counter with carry-in and carry-out. *This circuit is a transcription of the 74169 TTL design.*

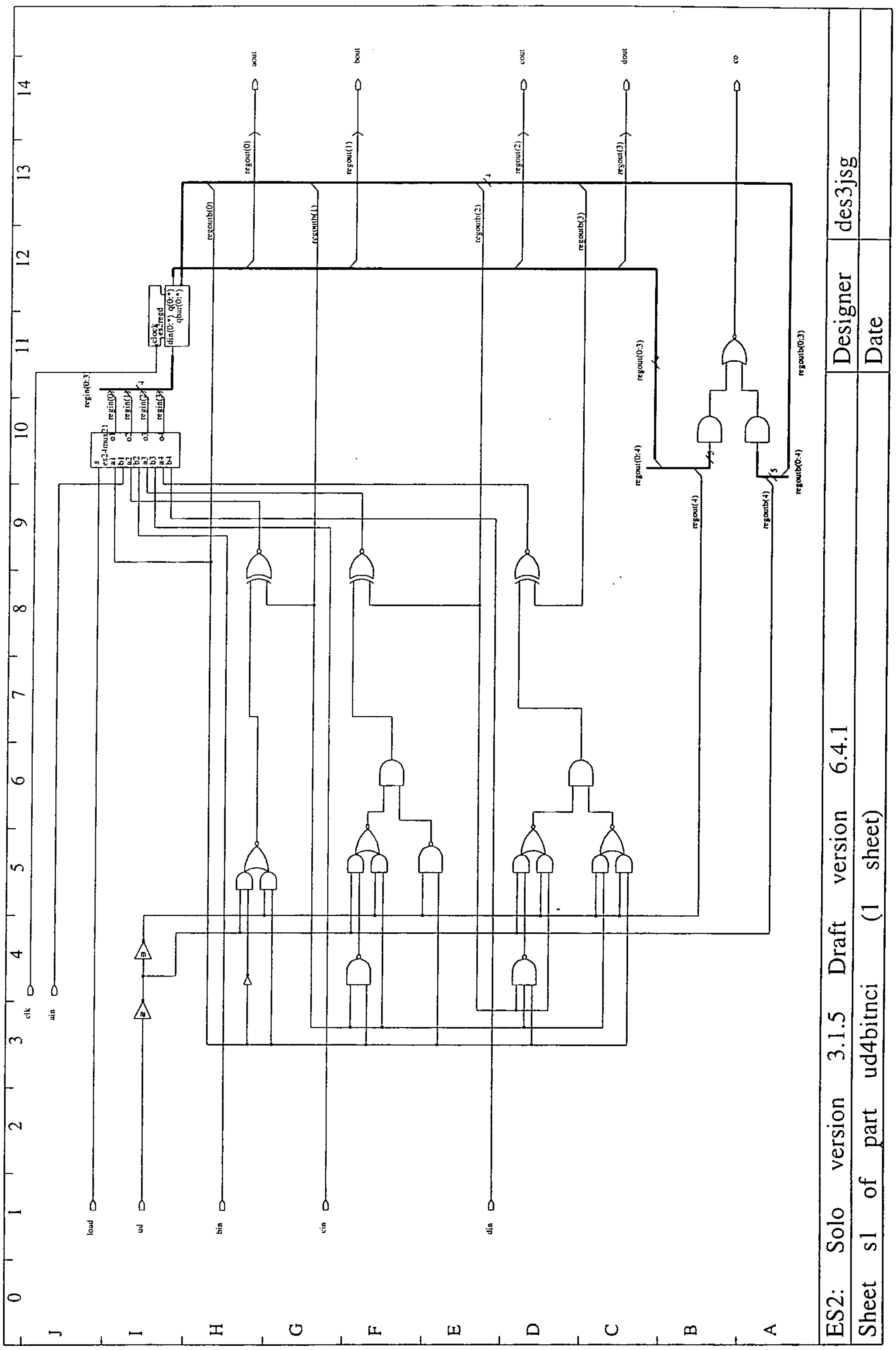


Figure 6.19: 4-bit counter with no carry-in. *This circuit is a transcription of the 74169 TTL design but the carry-in line and associated gating is removed. Compare this to Figure 6.18.*

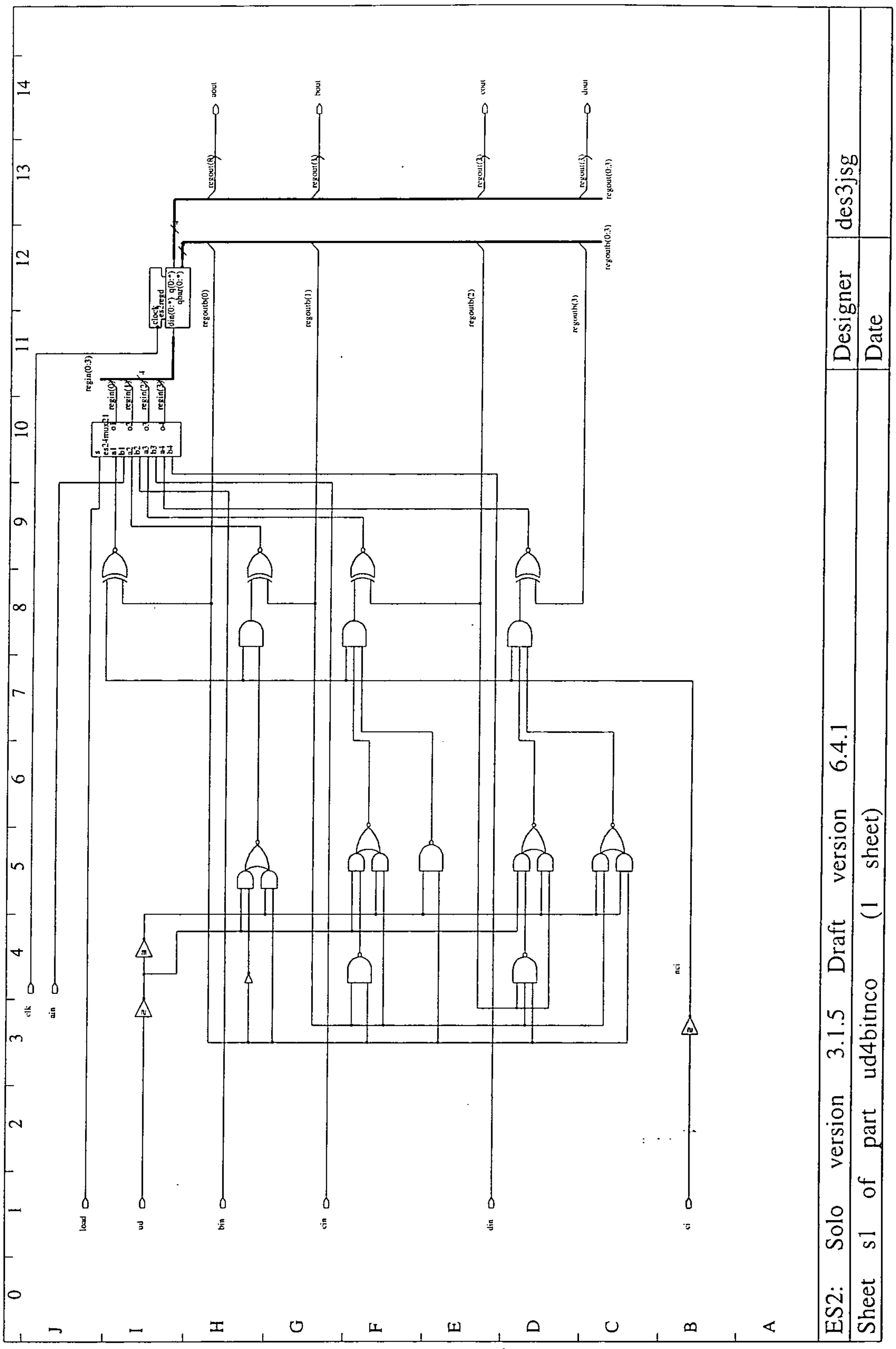


Figure 6.20: 4-bit counter with no carry-out. *This circuit is a transcription of the 74169 TTL design but the carry-out line and associated gating is removed. Compare this to Figure 6.18.*

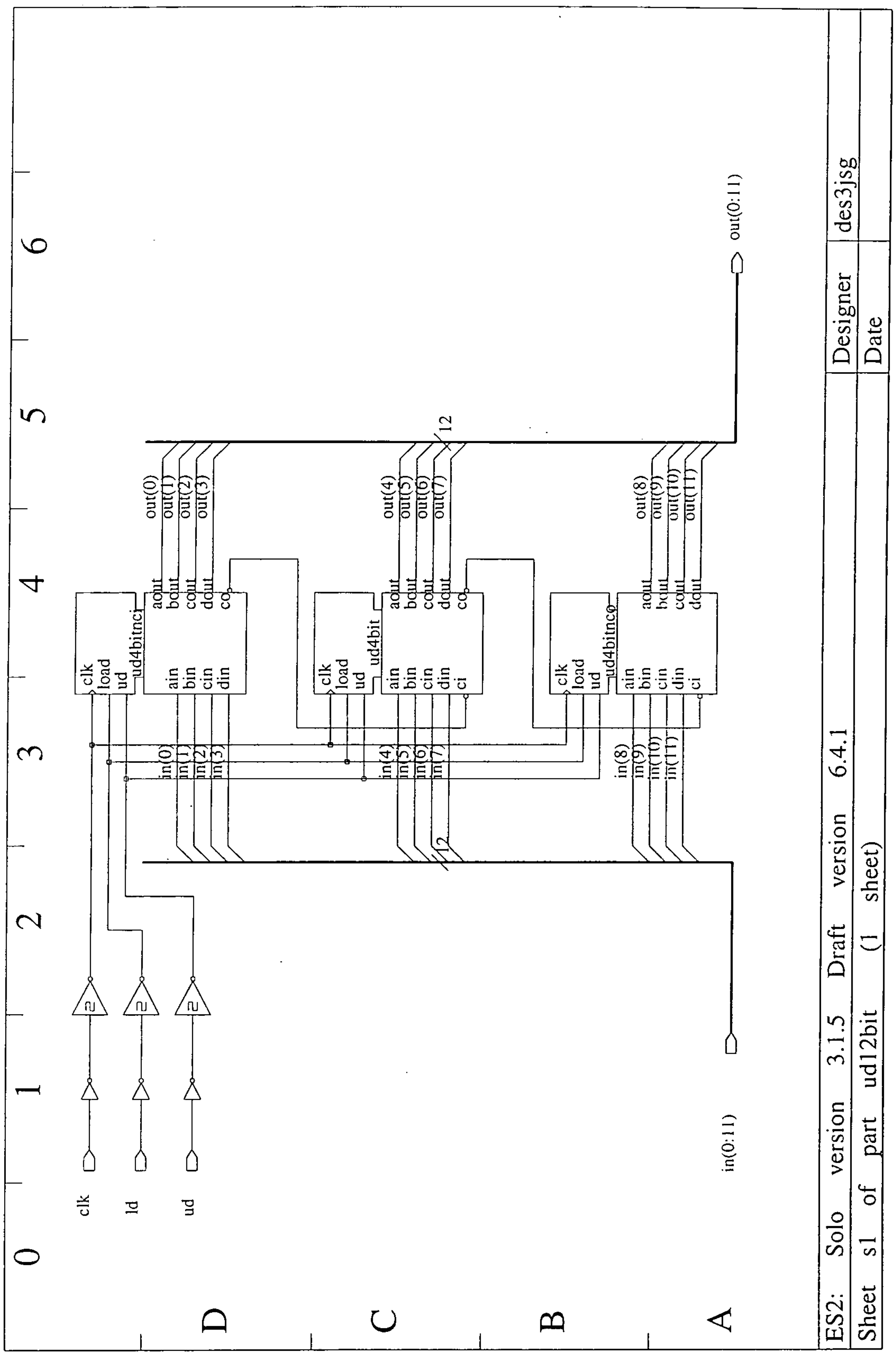

clk
ld
ud
in(0:11)
out(0:11)
ud4bitnci
ud4bit
ud4bitnco
12
ES2: Solo version 3.1.5 Draft version 6.4.1
Sheet s1 of part ud12bit (1 sheet)
Designer des3jsg
Date

Figure 6.21: 12-bit counter. *The counter is formed by cascading the 4-bit counters of Figure 6.18, Figure 6.19 and Figure 6.20. Cascading the three counter variants marginally reduces the component count and circuit interconnection required.*

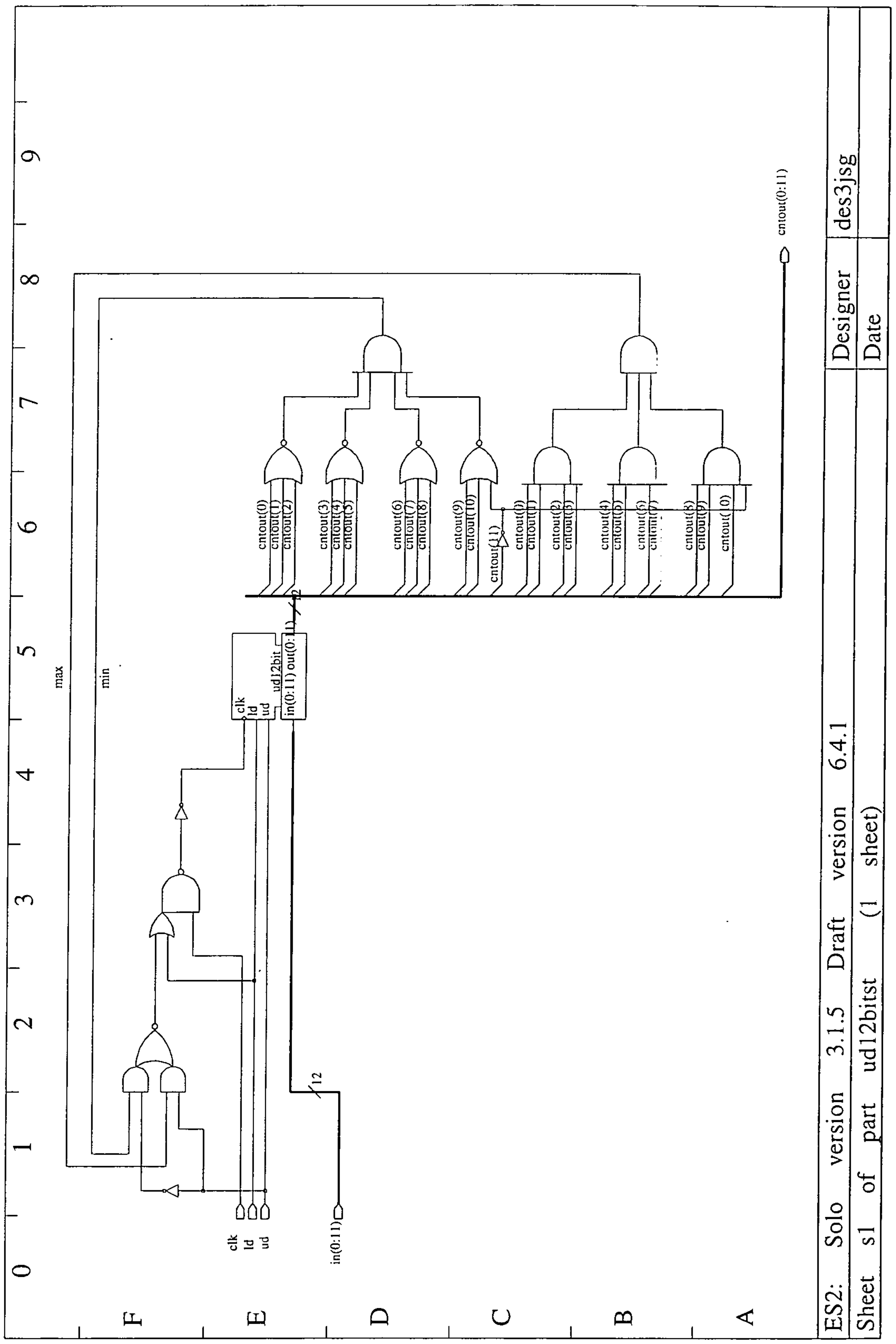


Figure 6.22: 12-bit counter with limit stops at -2048 and +2047. *The modular design enables the 12-bit counter to appear as a component around which the limit stop circuitry is configured.*

```
// ##########
// file:           ud12bitst.wdl
//
// description: Test and exercise 12-bit Up/Down counter with stops
// ##########

UP = 1;
DOWN = 0;
LOAD = 1;

void
rstCount(signal ld, signal inSig(11:0), signal clock)
{
  ld = LOAD;
  inSig = 0x000;
  clkPulse(clock);
  ld = !LOAD;
}  // end of function rstCount()

void
setCount(signal ld, signal inSig(11:0), signal clock, int value)
{
  ld = LOAD;
  inSig = value;
  clkPulse(clock);
  ld = !LOAD;
}  // end of function setCount()

main()
{
        Input in(11:0);
        Input ud;
        Input ld;
        Input clk;
        Output cntout(11:0);

        Set_Cycle(1000);

        in = 0x000;
        ud = UP;
        ld = !LOAD;
        clk = 1;
        rstCount(ld, in, clk);

// verify counter stops at max value, after 5 clks should be at max.
// immediately after direction changes should count down.
        setCount(ld, in, clk, 0x7FA);
        for (i = 0; i < 10; i++) {
                clkPulse(clk);
        }  // for i
        Toggle(ud);

// after this set of clockings should be back at 0x7FA
        for (i = 0; i < 5; i++) {
                clkPulse(clk);
        }  // for i

// verify counter stops at min value, after 5 clks should be at min.
// immediately after direction changes should count up/
        setCount(ld, in, clk, 0x805);
        for (i = 0; i < 10; i++) {
                clkPulse(clk);
        }  // for i
        Toggle(ud);

// after this set of clockings should be back at 0x805
        for (i = 0; i < 5; i++) {
                clkPulse(clk);
        }  // for i

}  // end of main()
```

Figure 6.23: `wdl` code for exercising up/down 12-bit counter. *The file tests the limit stops of the counter by loading values just below the limits and driving the counter to those limits. When the direction of count is reversed the counter should move away from the limits.*

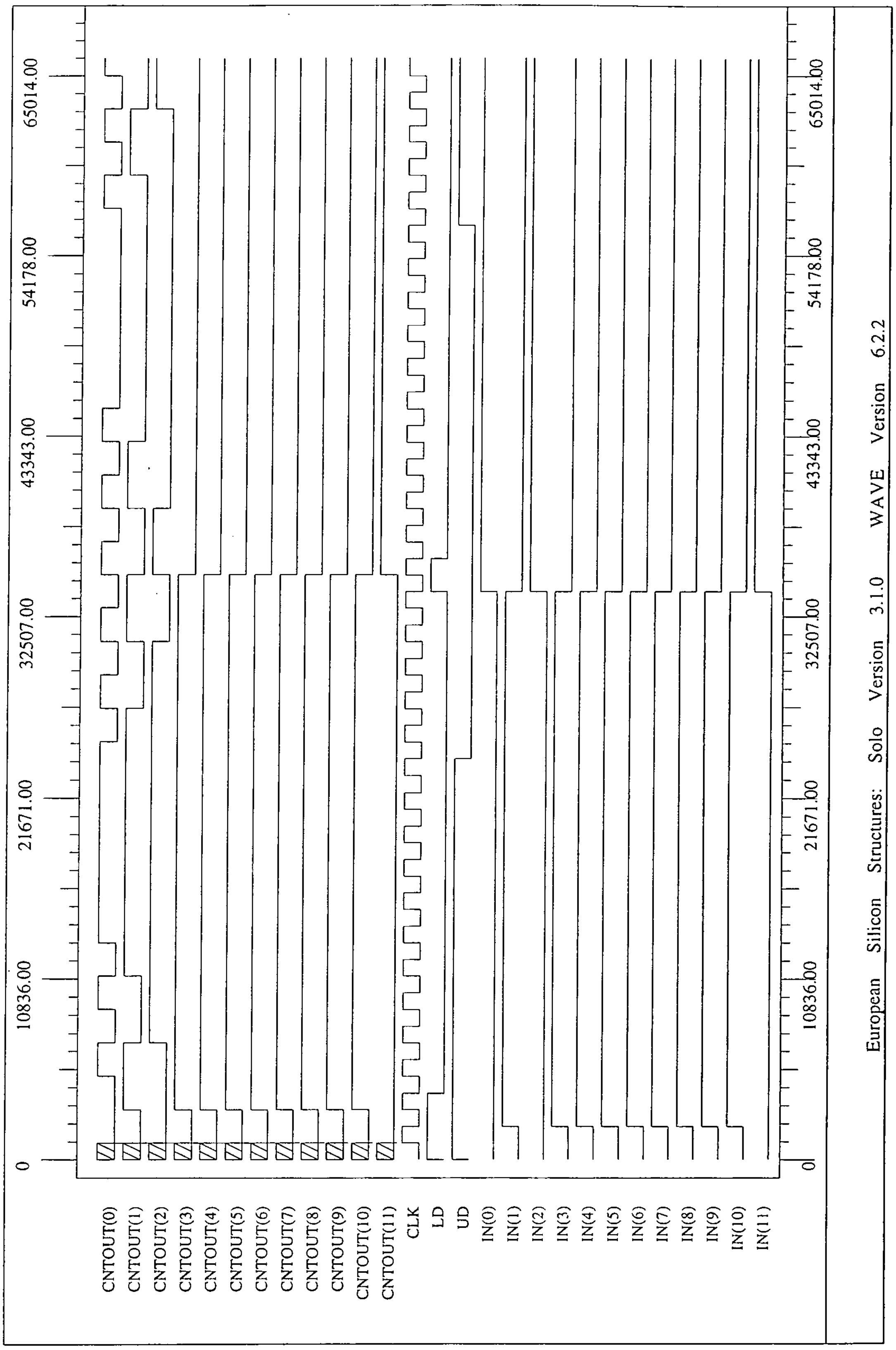


Figure 6.24: **wave** plot for an up/down 12-bit counter. *The first half of this plot demonstrates the halting of the counter at the upper limit, 0x7ff, while the second half demonstrates the counter halting at the lower limit, 0x800.*

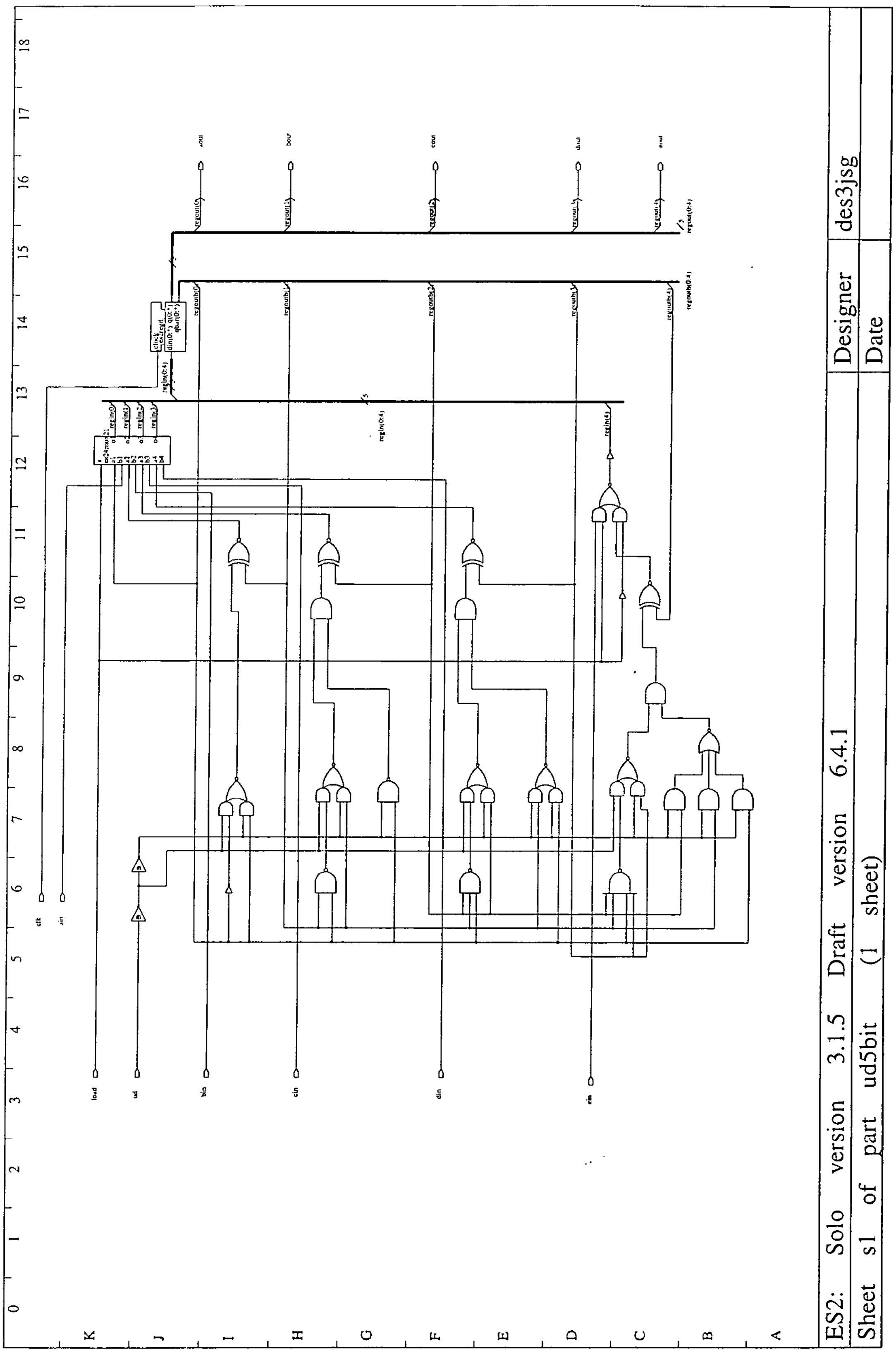

ES2: Solo version 3.1.5 Draft version 6.4.1
Sheet s1 of part ud5bit (1 sheet)
Designer des3jsg
Date
load
ud
bin
cin
din
ein
clk
ain

Figure 6.25: 5-bit counter with no carry-in or carry-out. *This circuit extends the method of the 4-bit counter, Figure 6.18, but has no carry lines associated with it.*

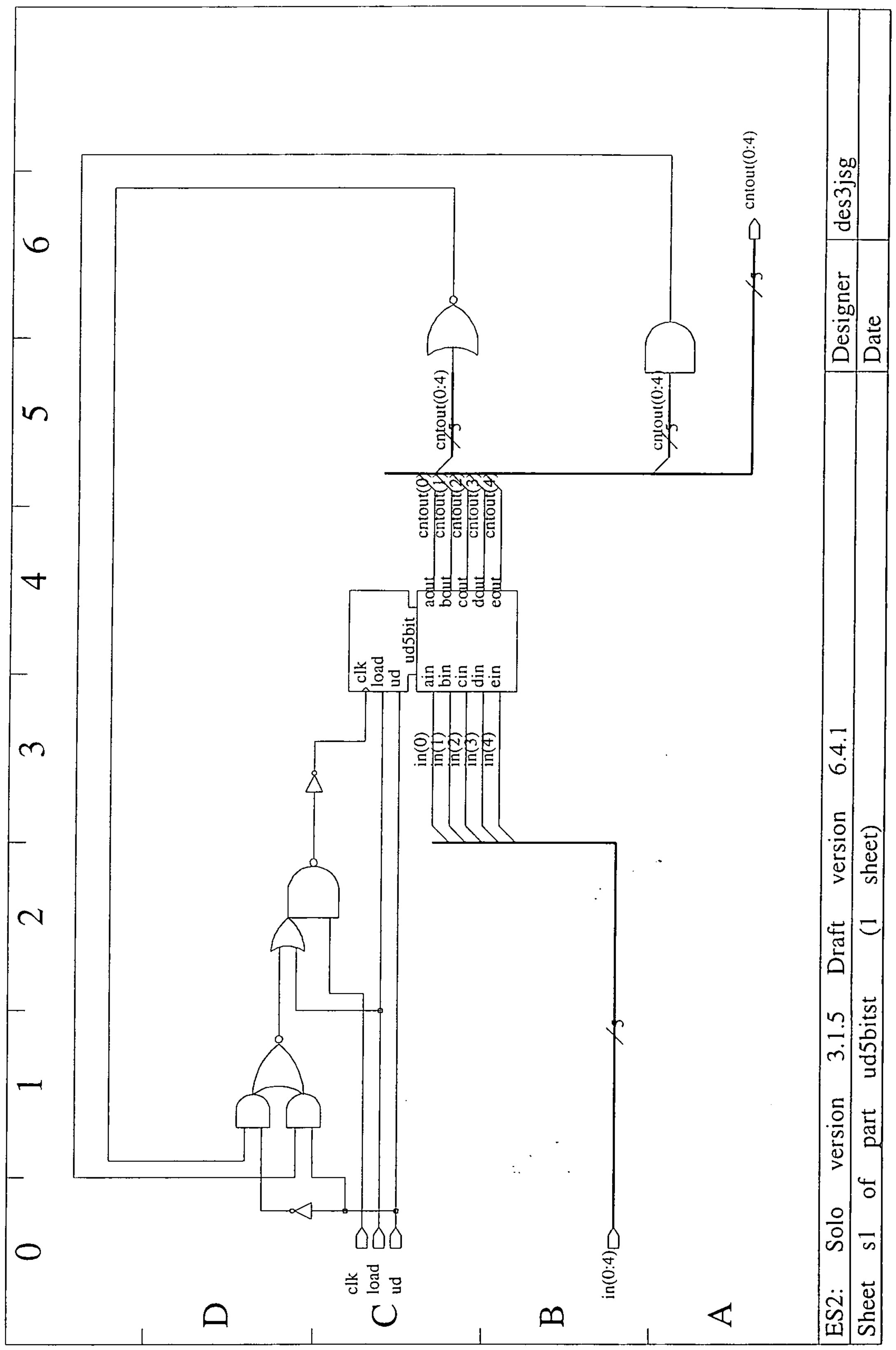


Figure 6.26: 5-bit counter with limit stops at 0 and +80. *As per the 12-bit design, the basic counter module has been augmented by the limit stop circuitry.*

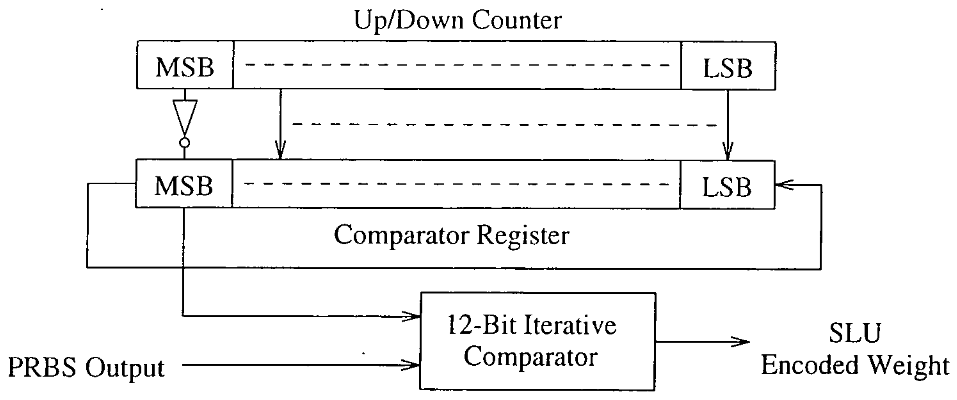


Figure 6.27: SLB weight encoding. *Every 12 cycles the contents of the Up/Down counter are transferred to the Comparator register. A comparison with a PRBS stream is performed to encode the counter value. Note the inversion of the MSB during the transfer to shift the counter value.*

```
{ ##########
{ part:           wghtenc
{
{ description:  Formed by manual extraction from a model file created by 'draft'
{ ##########
Part wghtenc [rng, in(0:11), rst, ud, ld, clk, en] → t,dout(0:11)

    Signal din(0:11)
    Signal bitwghtout
    Signal notclk
    Signal clkbuf
    Signal ldit
    Signal notrst
    Signal rstbuf
    Signal tmpt

    not [clk] → notclk
    buffer2 [notclk] → clkbuf
    not [rst] → notrst
    buffer2 [notrst] → rstbuf
    and [ld, en] → ldit
    comp_iter [clkbuf,rng,bitwghtout,rstbuf] → --,tmpt
    bdff [notrst,tmpt] → --, t
    es2sreg ps(12,2) [din(0:11),clkbuf,notrst] → bitwghtout,--
    ud12bitst [in(0:11),ud,ldit,clkbuf] → dout(0:11)
    not [dout(11)] → din(0)
    dout(10) → din(1)
    dout(9) → din(2)
    dout(8) → din(3)
    dout(7) → din(4)
    dout(6) → din(5)
    dout(5) → din(6)
    dout(4) → din(7)
    dout(3) → din(8)
    dout(2) → din(9)
    dout(1) → din(10)
    dout(0) → din(11)

End                                               { end of Part wghtenc declaration
```

Figure 6.28: `model` code for SLB input weight encoder. *The file specifies the circuit of Figure 6.27. Note the connection of the two buses, dout and din, has had to be done explicitly line by line.*

```
{ ##########
{ part:           demux5to17
{
{ description:  Decoder for selecting appropriate input weight
{               register for use.
{ ##########
Part demux5to17 [addr(0:4)] → select(0:16)

  Signal a0(0:3), a0bar(0:3), a1(0:3), a1bar(0:3)
  Signal a2(0:3), a2bar(0:3), a3(0:3), a3bar(0:3)
  Signal a4bar(0:3)

  cs2nbuff (4,2,0) [addr(0)] → a0(0:3)
  cs2nbuff (4,2,1) [addr(0)] → a0bar(0:3)
  cs2nbuff (4,2,0) [addr(1)] → a1(0:3)
  cs2nbuff (4,2,1) [addr(1)] → a1bar(0:3)
  cs2nbuff (4,2,0) [addr(2)] → a2(0:3)
  cs2nbuff (4,2,1) [addr(2)] → a2bar(0:3)
  cs2nbuff (4,2,0) [addr(3)] → a3(0:3)
  cs2nbuff (4,2,1) [addr(3)] → a3bar(0:3)
  cs2nbuff (4,4,1) [addr(4)] → a4bar(0:3)

  cs2and [a4bar(0),a3bar(0),a2bar(0),a1bar(0),a0bar(0)] → select(0)
  cs2and [a4bar(0),a3bar(0),a2bar(0),a1bar(0), a0(0)] → select(1)
  cs2and [a4bar(0),a3bar(1),a2bar(1), a1(0),a0bar(0)] → select(2)
  cs2and [a4bar(0),a3bar(1),a2bar(1), a1(0), a0(0)] → select(3)
  cs2and [a4bar(1),a3bar(2), a2(0),a1bar(1),a0bar(1)] → select(4)
  cs2and [a4bar(1),a3bar(2), a2(0),a1bar(1), a0(1)] → select(5)
  cs2and [a4bar(1),a3bar(3), a2(1), a1(1),a0bar(1)] → select(6)
  cs2and [a4bar(1),a3bar(3), a2(1), a1(1), a0(1)] → select(7)
  cs2and [a4bar(2), a3(0),a2bar(2),a1bar(2),a0bar(2)] → select(8)
  cs2and [a4bar(2), a3(0),a2bar(2),a1bar(2), a0(2)] → select(9)
  cs2and [a4bar(2), a3(1),a2bar(3), a1(2),a0bar(2)] → select(10)
  cs2and [a4bar(2), a3(1),a2bar(3), a1(2), a0(2)] → select(11)
  cs2and [a4bar(3), a3(2), a2(2),a1bar(3),a0bar(3)] → select(12)
  cs2and [a4bar(3), a3(2), a2(2),a1bar(3), a0(3)] → select(13)
  cs2and [a4bar(3), a3(3), a2(3), a1(3),a0bar(3)] → select(14)
  cs2and [a4bar(3), a3(3), a2(3), a1(3), a0(3)] → select(15)
  cs2and [ addr(4),a3bar(0),a2bar(0),a1bar(0),a0bar(0)] → select(16)

End                                             { end of Part demux5to17 declaration
```

Figure 6.29: `model` code for address selector/decoder. *Basic five line decoder, a 5-bit address is converted to one of 17 active output lines.*

```
{ ##########
{ part:           muxnto1
{
{ description:  An arbitary n-line to 1 multiplexor.
{ ##########
Part muxnto1 (elems) [in(0:elems-1),sel(0:elems-1)] → out

   Signal notsel(0:elems-1)

   Integer elemlp

   If elems < 5 Then
       Error "SOLO lib may exist for multiplexor size"
   Else
       For elemlp= 0 : (elems - 1) Cycle
           not [sel(elemlp)] → notsel(elemlp)
           tribuf1 [sel(elemlp),notsel(elemlp),in(elemlp)] → out
       Repeat                                                  { end For elemlp
   Endif                                                        { end If elems

End                                             { end of Part muxnto1 declaration.
```

Figure 6.30: `model` code for arbitrary N input multiplexor. *This circuit description iteratively builds a multiplexor of arbitrary size. Note how by use if the* If *statement feedback can be sent to the user to notify them of specific conditions.*

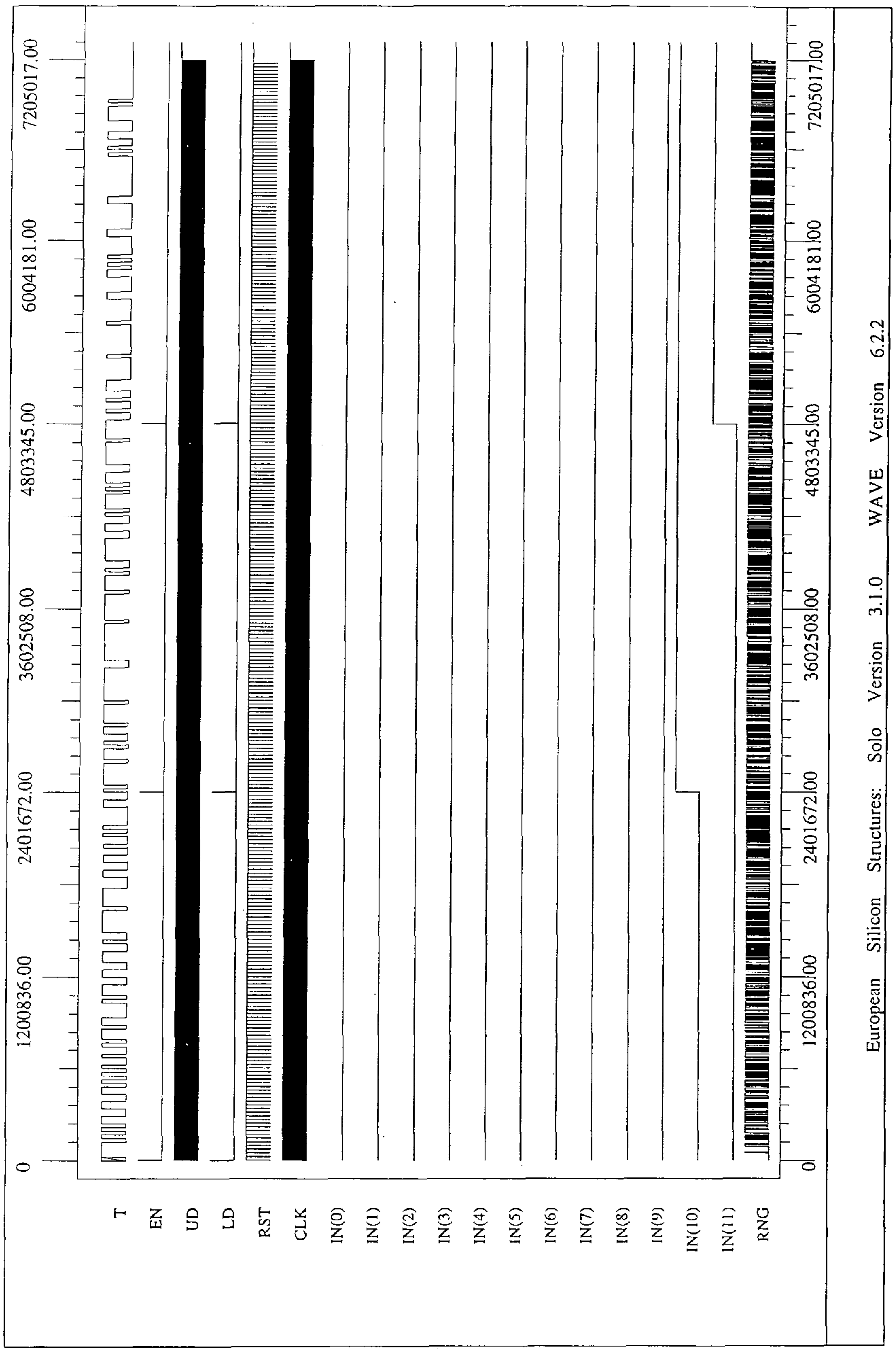


Figure 6.31: **wave** plot for input weight encoder performance. *The weight encoder is loaded with values corresponding to 0.5, 0.75 and 0.25. The output, T, can be seen to have an on period which corresponds to these conditions. The UD line is constantly toggled to maintain the counter at a stable value.*

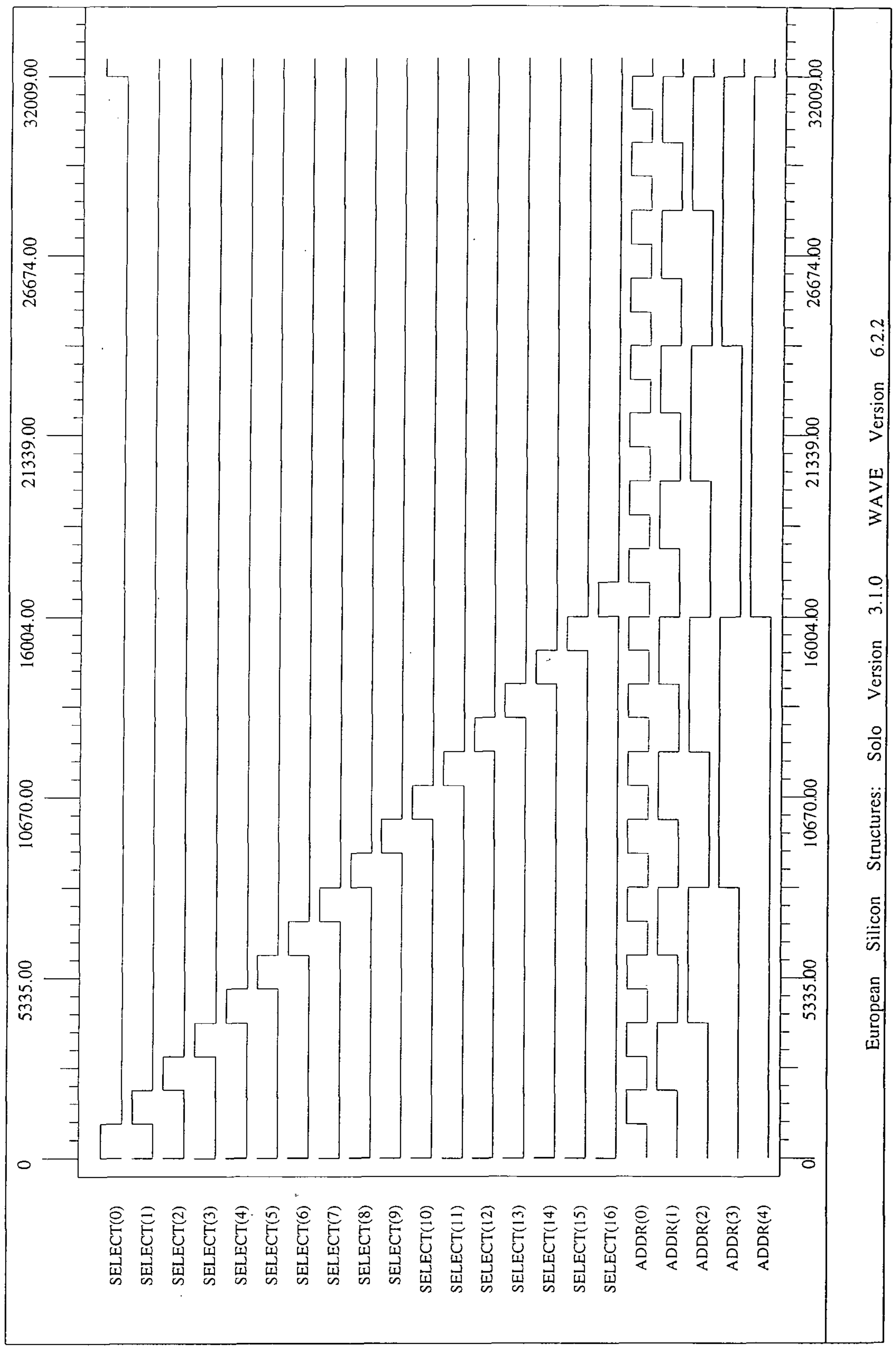


Figure 6.32: **wave** plot for demultiplexor/address decoder. *By counting up through the 32 address combinations an output on the correct address line occurs only for addresses 0 to 16.*

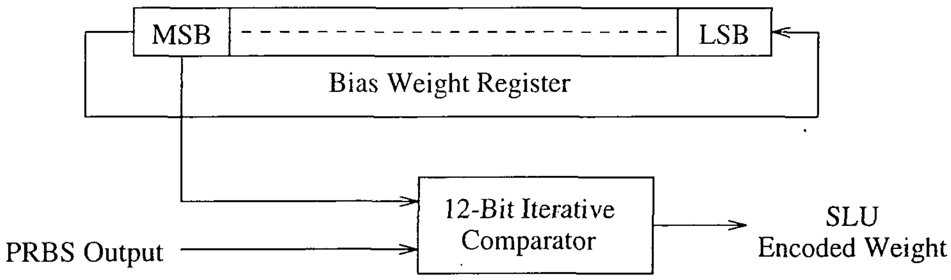


Figure 6.33: SLU weight encoding. *This circuit is similar to Figure 6.27 but without the 12-bit counter.*

```
{ ##########
{ part:           wght10
{
{ description:  Weight encoder for 1/10
{ ##########
Part wght10 [clk,rst,x] → t

        Signal y
        Signal tmpt
        Signal en
        Signal notclk

        not [clk] → notclk

        not [rst] → en
        es2sreg ps (12,2) [Gnd,Gnd,Gnd,Vdd,Vdd,Gnd,Gnd,Vdd,Vdd,Gnd,Vdd,Gnd,clk,en] → y,--
        comp_iter [notclk,x,y,rst] → --,tmpt
        bdff [en, tmpt] → --,t

End                                                      { end of Part wght10
```

Figure 6.34: `model` code examples of a static SLU encoder. *Observe how the shift register can always be re-initialised to the same value since the load inputs are tied to either Vdd or Gnd.*

```
{ #########
{ part:          divide_cell
{
{ description:  Building block for N pulse divider
{ #########
Part divide_cell [in,prev] → out,next

    Signal notin

    and [in,prev] → out
    not [in] → notin
    and [prev,notin] → next

End                                                     { end of Part divide_cell declaration
```

Figure 6.35: `model` code for divide cell building block. *This building block can be seen repeatedly in Figure 4.12.*

```
{ #########
{ part:          n_pulse_div
{
{ description:  N pulse divider for input to stochastic summer
{ #########
Part n_pulse_div (streams) [in(2:streams)] → out(1:streams)

    Integer streamlp

    Signal prev(1:streams)
    Signal notstream2

    If streams<2 Then
        Error "Too few pulse streams specified"
    Else
        wire[in(streams)] → out(streams)
        not [in(streams)] → prev(streams)

        For streamlp=(streams - 1):2 By -1 Cycle
            divide_cell [in(streamlp),prev(streamlp + 1)] →
                out(streamlp),prev(streamlp) : divide_cell(streamlp)
        Repeat                                                  { end For streamlp

        prev(2) → out(1)
    Endif                                                        { end If streams

End                                                     { end of Part n_pulse_div declaration
```

Figure 6.36: `model` code for complete N pulse divider. *This is another parameterised circuit enabling arbitrary long pulse divide trees to be produced from the divide_cell block of Figure 6.35.*

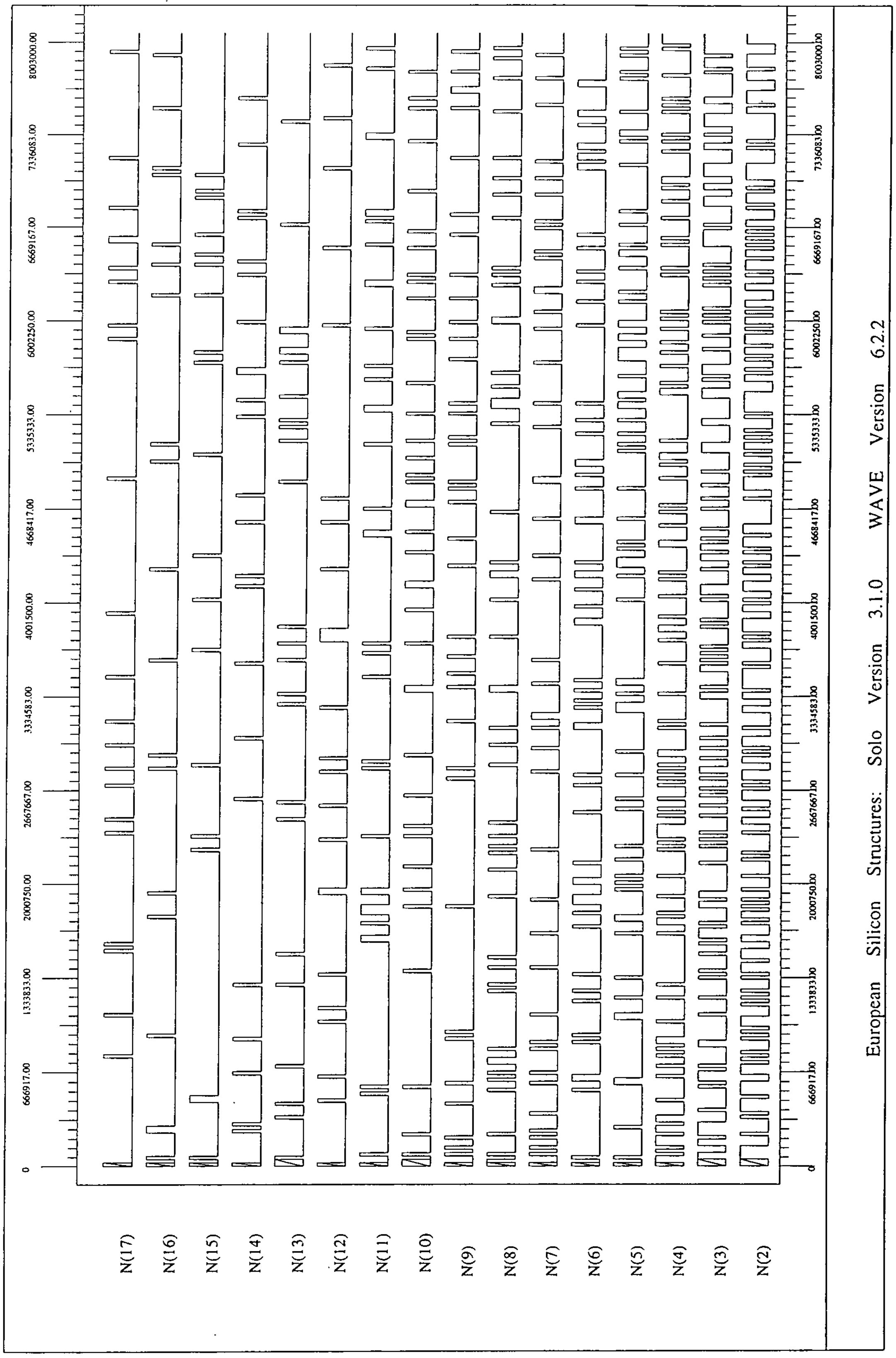


Figure 6.37: **wave** plot demonstrating static weight encoding. *The time a line is high can be seen to increase progressively from N(17) to N(2).*

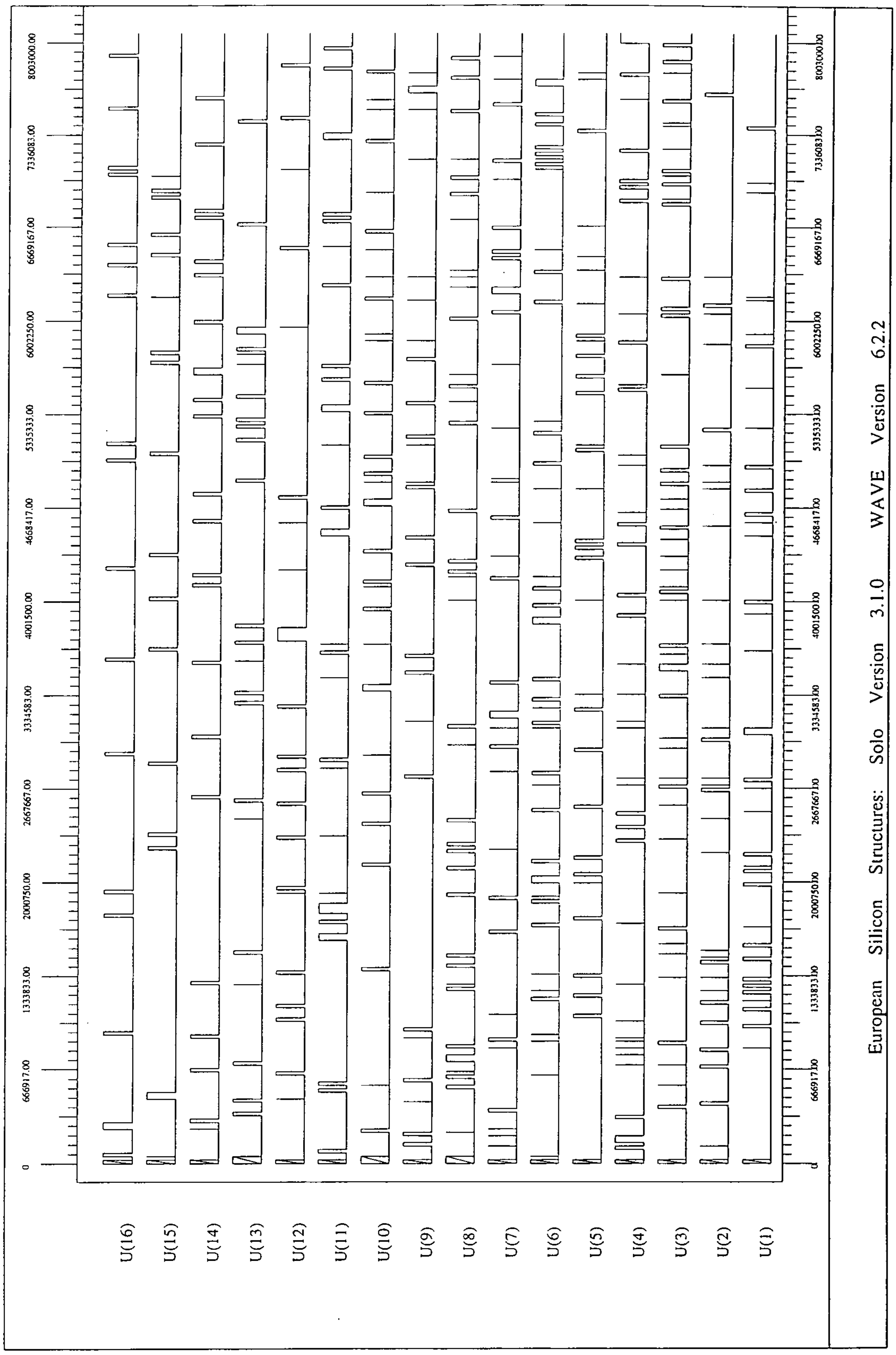


Figure 6.38: **wave** plot demonstrating the $\frac{1}{17}$ gating streams. *Each of the 17 bus lines has a probability of $\frac{1}{17}$ of being high. The spikes are filtered out by the latching of the U values.*

```
{ ##########
{ part:            slbip_mul_block
{
{ description:  Single Line Bipolar Multiplier block. Multiplies two
{                 buses of signals x(0:elems) and w(0:elems) by use of
{                 xor gates.
{                 Input parameter 'elems' the number of multipliers - 1
{                 there will be.
{ ##########
Part slbip_mul_block (elems) [x(0:elems-1), w(0:elems-1)] → xw(0:elems-1)

    Integer elemlp

    If elems<1 Then
        Error "No elements to multiply"
    Else
        For elemlp =0:elems-1 Cycle
            eqv [x(elemlp),w(elemlp)] → xw(elemlp) : xor(elemlp)
        Repeat                                                    { end For elemlp
    Endif                                                          { end If elems

End                                          { end of Part slbip_mul_block declaration
```

Figure 6.39: model code for SLB multiplication of input values and weights. *This circuit is simply an array of* **XOR** *gates.*

```
{ ##########
{ part:            sluni_mul_block
{
{ description:  Single Line Unipolar Multiplier block, Multiplies two
{                 buses of signals x(0:elems) and w(0:elems) by use of
{                 and gates.
{                 Can also be used for the gating in a Multiple Input
{                 Summer.
{                 Input parameter 'elems' the number of multipliers - 1
{                 there will be.
{ ##########
Part sluni_mul_block (elems) [x(0:elems-1), w(0:elems-1)] →
                              xw(0:elems-1)

    Integer elemlp

    If elems<1 Then
        Error "No elements to multiply/gate"
    Else
        For elemlp=0:elems-1 Cycle
            and [x(elemlp),w(elemlp)] → xw(elemlp) : and2(elemlp)
        Repeat                                                    { end For elemlp
    Endif                                                          { end If elems

End                                          { end of Part sluni_mul_block declaration
```

Figure 6.40: model code for SLU multiplication/gating of weighted inputs. *This circuit is simply an array of* **AND** *gates. One input to each* **AND** *gate is the weighted input, the second is a* $\frac{1}{17}$ *gating signal.*

```
{ ##########
{ part:            gaus1
{
{ description:  Produce Gaussian numbers
{ ##########
Part gaus1 [sumin,prbs,clk,rst] → t

      Signal notrst
      Signal eseqbase(0:79)
      Signal eseqout, noteseqout
      Signal inc, dec, notdec
      Signal countout(0:4)
      Signal reg12out
      Signal tmpt
      Signal rst80, rst12
      Signal clk80, notclk80
      Signal clk12st, notclk12st
      Signal ud5rst
      Signal notclk

      Signal tmp, tmp2, tmp1, tmpinc

      Gnd → eseqbase(0)
      .
      .
      .
      Vdd → eseqbase(79)

      not [clk80] → notclk80
      count80 [clk,rst] → clk80
      and [rst,notclk80] → rst12
      count12 [notclk,rst12] → clk12st
      not [rst] → notrst
      es2sreg ps (80,2) [eseqbase(0:79),clk,notrst] → eseqout,--

      and [prbs,eseqout] → inc
      or [notrst,clk80] → ud5rst

      not [clk] → notclk
      or [prbs,ud5rst] → tmp
      and [tmp,notclk] → tmp2
      or [inc,ud5rst] → tmpinc
      ud5bitst [Gnd,Gnd,Gnd,Gnd,Vdd,tmpinc,ud5rst,tmp2] → countout(0:4)

      es2sreg ps (12,2) [countout(4),countout(3),countout(2),countout(1),countout(0),Gnd(0:6),clk,ud5rst] →
reg12out,--
      comp_iter [clk,reg12out,sumin,clk12st] → --,tmpt

      bdff [notclk80,tmpt] → t,--

End                                                    { end of Part gaus1 declaration
```

Figure 6.41: `model` code for sigmoidal transformation circuit. *This circuit produces Gaussian distributed random numbers which the weighted sum of products is compared. This performs the sigmoid transform.*

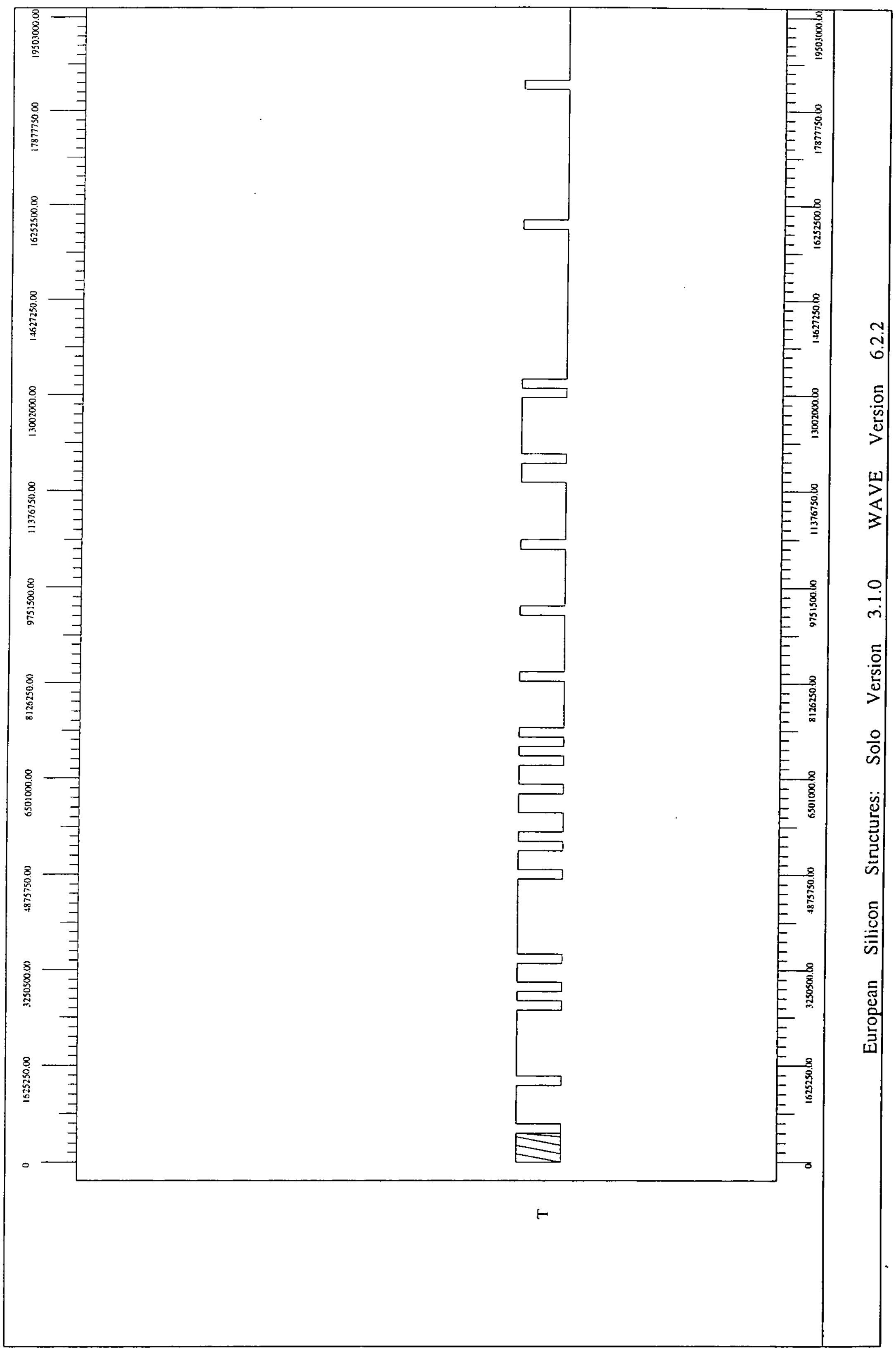


Figure 6.42: **wave** plot demonstrating testing of sigmoidal transform. *Due to the omission of a single inverter, as the input values increase from* 0.2 $\rightarrow$ 0.5 $\rightarrow$ 0.8 *the output, T, becomes less dense rather than more dense, but the appropriate non-linear mapping does exist.*

```
{ ##########
{ part:            neur
{
{ description:  The neuron.
{ ##########
Part neur [clk,in(0:15),addr(0:4),ud(0:16),rw,rst] → out,init(0:11),sumout,outwght(0:16)

   Signal prbsout(1:27)
   Signal rng(1:34)
   Signal sumin_17(0:16)
   Signal sum
   Signal clk_in
   Signal rst_in
   Signal rw_in
   Signal rwbuf(0:2)
   Signal in_in(0:15)
   Signal init_in(0:11), notinit_in(0:11)
   Signal addr_in(0:4)
   Signal ud_in(0:16)

   Signal wghtout(0:11)
   Signal wghtouta(0:11)
   Signal out_in
   Signal wghtenc_in(1:16)

   Signal clk12a, notclk12a
   Signal rst12

   Signal clkbuf(0:5)
   Signal rstbuf(0:4)

{
{ Pad connections omitted
{

   notarray (12) [notinit_in(0:11)] → init_in(0:11)

   es2nbuff (3,4,0) [rw_in] → rwbuf(0:2)
   es2nbuff (5,3,0) [rst_in] → rstbuf(0:4)
   es2nbuff (6,3,0) [clk_in] → clkbuf(0:5)

   count12a [clkbuf(0),rstbuf(4)] → clk12a
   not [clk12a] → notclk12a
   and [rstbuf(4),notclk12a] → rst12

   prbs27 [clkbuf(1),rstbuf(0)] → prbsout(1:27)
   prbs27to38 [prbsout(1:27)] → rng(1:34),--,--,--,--
   inpwght [clkbuf(2),in_in(0:15),init_in(0:11),rng(18:34),ud_in(0:16),addr_in(0:4),rw_in,rstbuf(1),rst12] →
sumin_17(0:16),wghtouta(0:11),wghtenc_in(1:16)

   es2regd (12) [clkbuf(3),wghtouta(0:11)] → wghtout(0:11),--,--,--,--,--,--,--,--,--,--,--,--
   summer16 [sumin_17(0:16),rng(2:17),rstbuf(2),rst12,clkbuf(4)] → sum
   gausl [sum,rng(1),clkbuf(5),rstbuf(3)] → out_in

End                                                            { end of Part neur declaration
```

Figure 6.43: Basic `model` code for the complete neuron. *Due to the modular nature of the design process the final circuit is a concise description of the design.*

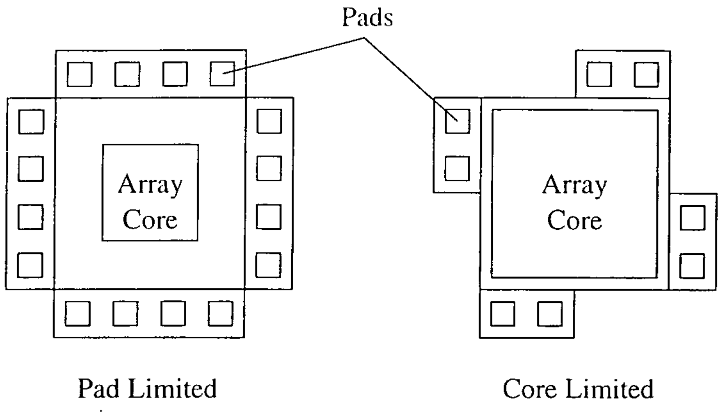


Figure 6.44: Example of pad and core limited designs. *A pad limited design is one where the limiting factor on size is dominated by the number of pads which must enclose the circuit. For a core limited design the basic circuitry has the most influence on eventual size.*

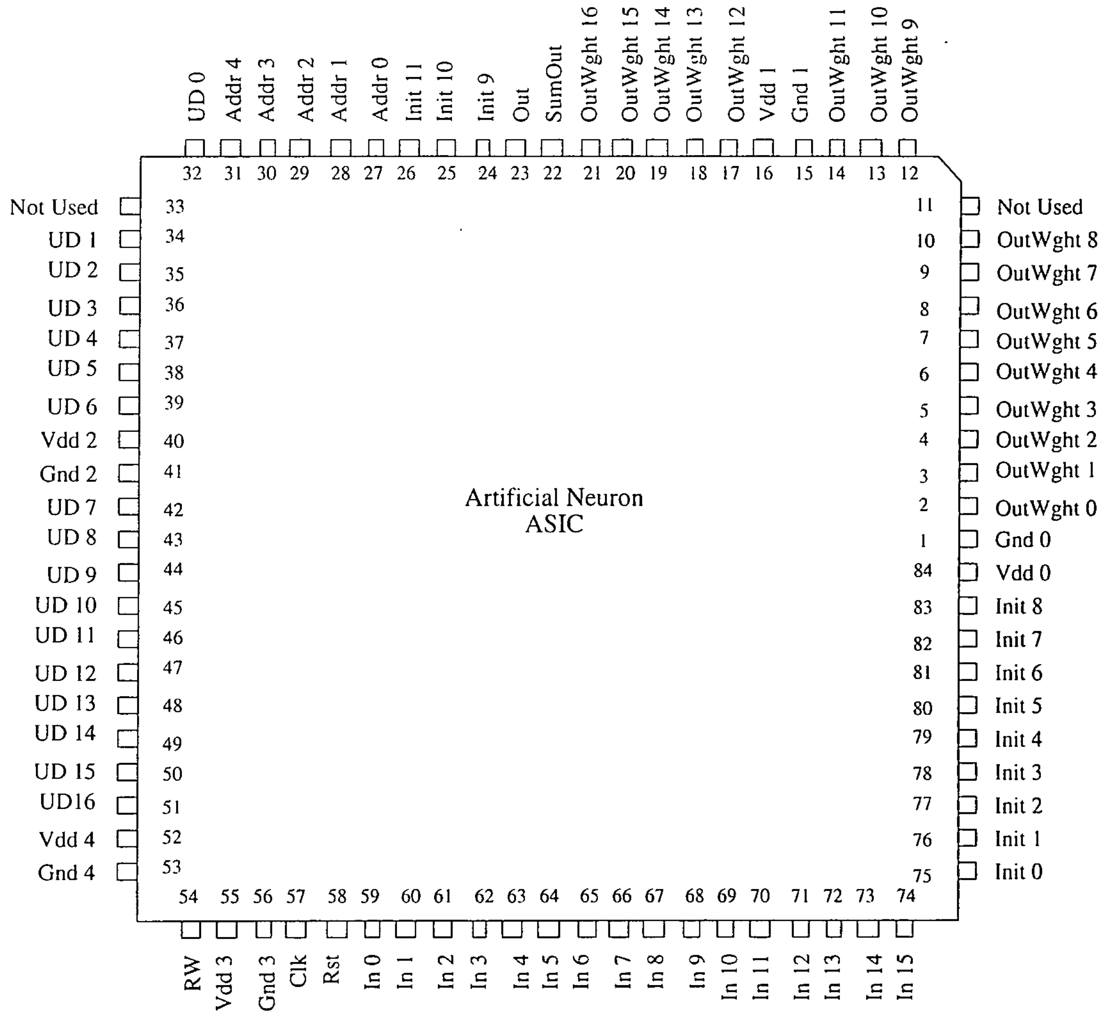


Figure 6.45: Neuron ASIC pin configuration. *In addition to the necessary input/output connections, unused pins are connected to monitor key points in the system, OutWght and SumOut.*

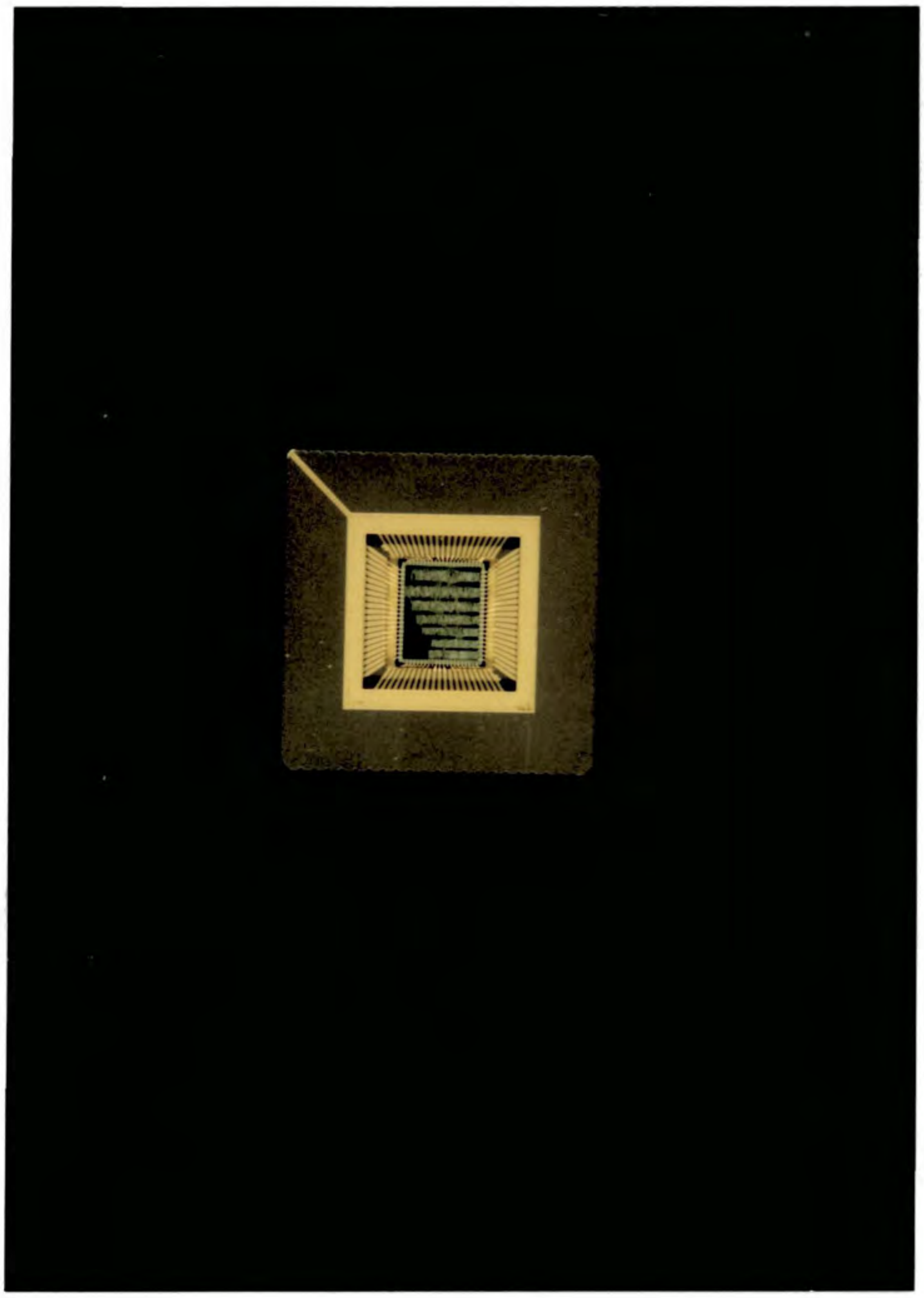


Figure 6.46: A photograph displaying the resulting fabricated neuron.

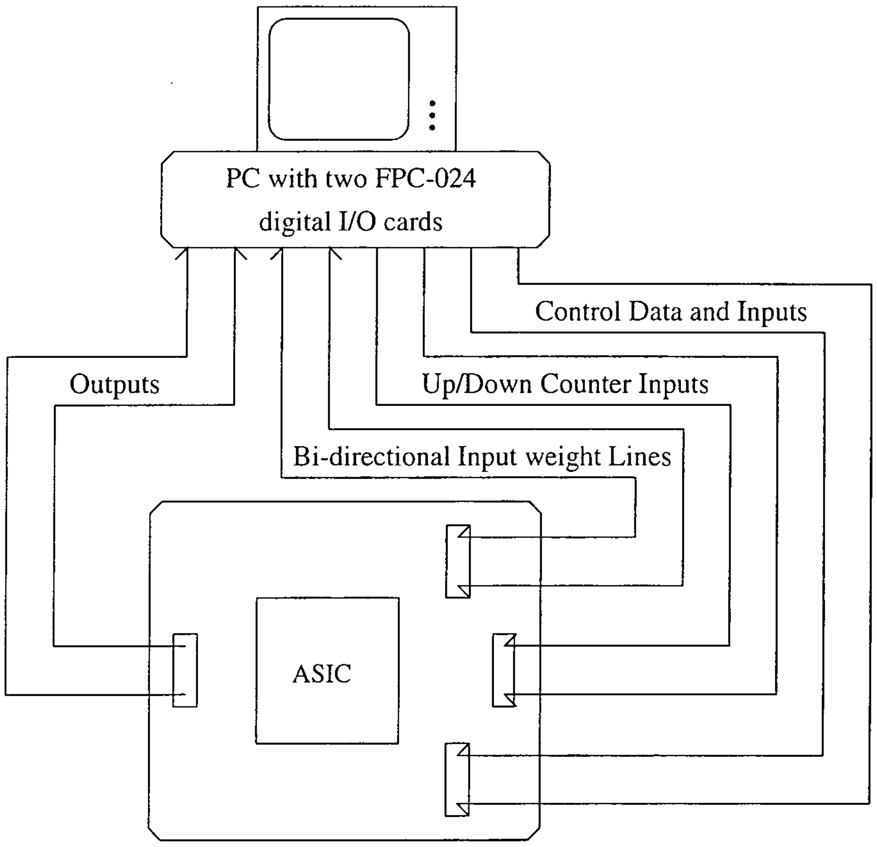


Figure 6.47: Neuron ASIC hardware test configuration. *The ASIC pin connections are broken out into four sets of lines which are controlled and monitored via two FPC-024 digital I/O cards mounted in a PC. This approach exchanges hardware complexity for software complexity.*

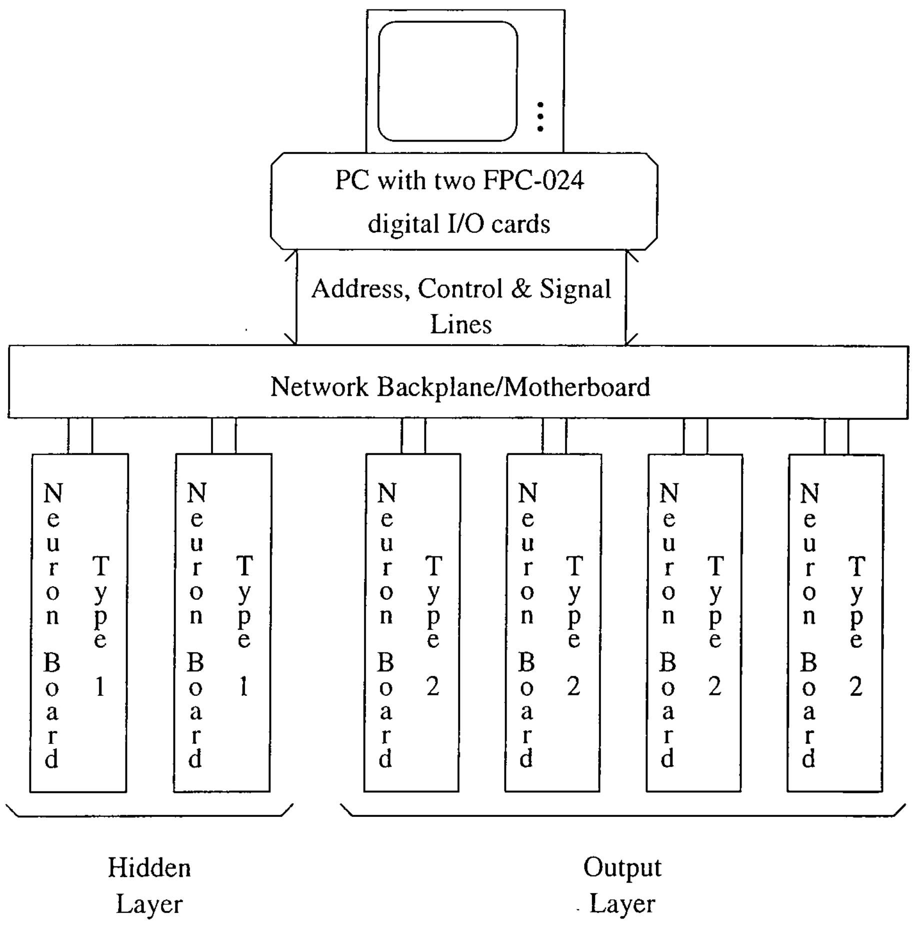


Figure 6.48: Full 4–2–4 Hardware Neuron System.

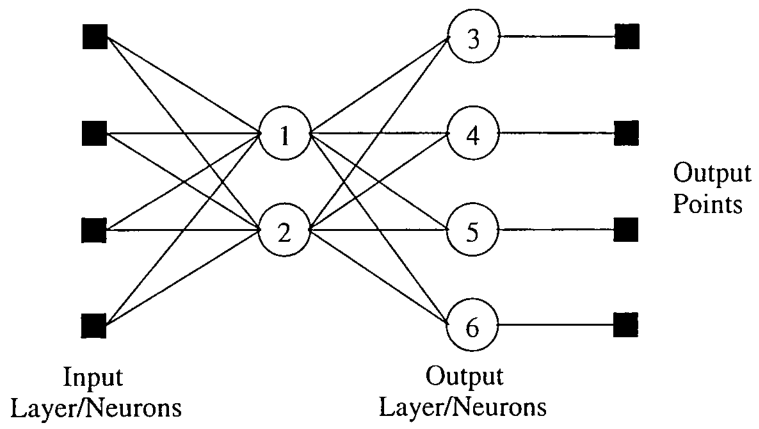


Figure 6.49: 4–2–4 Feedforward neural system. *The number of each neuron signifies the order it was multiplexed through the single neuron device.*

Chapter 7

Conclusions and Further Work

7.1 Conclusions

In this thesis the development of a hardware neuron operating by the use of stochastic pulse rate encoding principles has been undertaken. The study of ANNs and algorithms is currently a wide area of research with much of the work conducted in the engineering field through software models and simulations. This is a slow process since ANNs are inherently a parallel method of information processing based upon many simple processing elements operating simultaneously. Software models and simulations will not usually be able to take full advantage of this process. The investigation and development of suitable hardware implementations of artificial neurons with the ability to adapt and train as they operate is a key area for research. The hardware realisation will enable the parallel power and speed of computation possible with such systems to be more fully realised.

The work described in this thesis initially focused upon appropriate architectures and algorithms for NNs suitable for transition into a hardware implementation. A critical review was conducted in Chapter 2 which highlighted some of main NN architectures with their algorithms describing why they may be of interest. The work of Barto *et al* into simple reinforcement learning using A_{R-P}, which are related to MLPs and backpropagation, was shown to be of particular interest with its ability to assign credit and enable a network to adapt to solve a problem especially when hidden layer processing elements behaved stochastically. The methods proposed by Barto *et al* were validated by their application to two standard test problems, the encoder/decoder and the exclusive-OR. It was shown that the algorithm could enable a feedforward network to adapt to a solution. It was also shown that a punishment signal in the credit assignment term was important for the best results to be obtained. The A_{R-P} reinforcement strategies have been noted to be particularly interesting due to their comparative ease of transition into a hardware implementation.

The A_{R-P} reinforcement schemes have been extended to produce two new models, the

Q-model A_{R-P} and the T-model A_{R-P} which build upon the P-model A_{R-P} and S-model A_{R-P} strategies of Barto *et al.* These two new systems use the same single reinforcement signal for all neurons in the network and the output neurons in the network now behave stochastically. These new models have been demonstrated to work for the test problems of a 4–2–4 encoder/decoder and an **XOR** problem. The scalability of these global A_{R-P} training strategies to a larger network of neurons is an issue which must be addressed since it did not prove possible to train an 8–3–8 encoder/decoder network using either of the two new strategies in the time allocated for training. It may be possible by investigation of the gain and asymmetry parameters of the adaption algorithm to overcome this potential scalability issue. These two new schemes do however have the potential to be simpler to implement in hardware than the original strategies of Barto *et al* from which they are developed.

A critical review of hardware implementation issues is conducted in Chapter 3 with the assessment of analogue and digital techniques for the formation circuits appropriate to ANNs. The fields of pulse rate encoding, both deterministic and stochastic, are shown to be attractive for ANN implementation. Stochastic pulse rate encoding is shown to be of practical interest due to the efficiency and small size of its computational elements and its robustness to noise.

To be able to design a hardware artifical neuron based upon stochastic pulse rate encoding principles a knowledge of the circuits and their operation is required. A description of stochastic pulse rate encoding strategies, SLU, DLB and SLB, is provided in Chapter 4. In addition the following three novel circuits are developed.

A Novel Subtracter

A modification of the original addition circuits presented by Leaver enables subtraction to be performed between two lines when signals are encoded using SLU strategies, §4.5.1. The circuit operates by the removal of pulses from one signal line commensurate with the pulses present on a second signal line.

An N-input Adder

This new N-input adder presented in §4.4.1 enables the addition of N equally weighted stochastic signals. The adder relies upon the generation of N equally weighted stochastic signals of value $\frac{1}{N}$ for which an extendable process and architecture are presented to achieve the task. The weighting signals are stochastic in nature and have the property that none of the weighting signals have coincident pulses, a necessary condition to enable accurate weighted summation.

Sigmoidal Transform Generator

Several possible techniques for the generation of a sigmoidal transform were considered in section 4.7, all but one were discounted as being difficult to realise in practice. The generation of a sigmoid by use of a Gaussian random number generator was pursued further. An E-sequence method is presented for the generation of Gaussian random numbers when operating in the stochastic pulse rate encoded domain. Using this technique the properties of the sigmoidal transform are adjusted by varying the Gaussian random number distribution. The gradient of the sigmoid can be varied by adjusting the variance of the distribution and the mid-range point adjusted by varying the mean of the Gaussian distribution.

For the encoding of signals into the stochastic pulse rate domain a supply of noise or random numbers is necessary. Chapter 5 develops the principle of the generation of multiple random numbers from a single PRBS generator. Given that the sequence of random numbers from a PRBS is sufficiently long it has been demonstrated that it is possible to generate multiple random numbers from a single sequence by taking a delayed tap-off of values for the sequence.

Results are presented demonstrating the suitability of the optimisation techniques of simulated annealing and genetic algorithms to the problem of optimising multiple tap-off combinations for multiple PRBS sequences. The tap-off combinations are optimised to produce even loading on the PRBS register elements, minimum number of total taps and the minimum deviation from the optimum distribution of delays between sequences.

Having discussed and demonstrated all the constituent elements for an artificial neuron operating using stochastic pulse rate encoding, a complete design for such a neuron is presented in Chapter 6. The design operates entirely within the stochastic pulse rate environment producing a function of a weighted sum of 16 inputs. The weights associated with each input have the ability to be adjusted on-line by means of either an up/down signal or by loading with a completely new value by external intervention.

The neuron circuit has been fabricated in 1.5μ technology using standard cells for the circuit components and demonstrated to operate. The sigmoidal transform does not produce as good a sigmoid as expected, but this can be attributed to the limited dynamic range of the underlying Gaussian distribution. A potential problem of a reduced dynamic range of output values was identified for the neuron when fewer than the maximum of 16 input to the neuron were utilised, §6.5.

The computational capability of a network of the fabricated neurons to perform a simple test task, the 4–2–4 encoder/decoder, has been demonstrated by multiplexing the operation of a network through a single device. The original implementation using a full network of six devices was set aside due to a problem in continuity between the fabricated device and its surrounding socket. To be able to perform off-line training of the network and then loading the learned weights into the neuron it was found that an accurate model of the neuron is required, in particular the sigmoid transform.

It has been found overall that, in general, the system of stochastic pulse rate encoded computation for the hardware realisation of an artificial neuron are a feasible and an attractive option due to the rapid rate of computation, the immunity to noise, efficiency of the circuits and the ability to adjust the weights on-line as the system operates. A potential A_{R-P} reinforcement learning strategy for amalgamation with the hardware has been identified to be worthy of consideration. However, the overhead of the supporting circuitry for weight storage and encoding, the currently poor sigmoid transform and the reduction of the output dynamic range must be considered in using this approach as it currently exists.

7.2 Further Work

The application of stochastic pulse rate computation techniques to the hardware realisation of an artifical neuron together with a reinforcement technique have demonstrated a good potential for further research and development in the field of ANNs. Several interesting areas of work can be identified which include the following.

New N-input Adder

The original N-input adder suffered from a reduction in the dynamic range of the output when less than the full N input lines were being used. To overcome this problem a divider tree for the N-input adder is necessary such that unused lines can be turned off. This will produce as adder which can sum any number of input lines $M : M \leq N$ with weighting factor $\frac{1}{M}$. Such a circuit is possible and is displayed in Figure 7.1.

For the new circuit for the generation of M pulse streams of value $\frac{1}{M}$ the appropriate $S_{N-x} \equiv S_M$ line is set high for the chain. This sets all outputs above it to low and the feed into the pulse multiplier below to high.

This new N-input adder could thus be implemented and used in the hardware design. External addressing would be necessary to turn on the appropriate select lines.

Improved Sigmoid Transform

Investigation into new alternative techniques for the generating the sigmoid transform could be conducted. Two possible approaches may be taken, the first is to find an alternative system for the formation of Gaussian random numbers and the second is to adopt a completely new approach. A new suggestion, therefore, is to create a piece-wise linear model of the sigmoid transform. A deterministic mapping is made from the decoded sum of weighted inputs through the sigmoid transform to be re-encoded stochastically. An additional attraction of this second option is that by adjusting the piece-wise linear model the characteristic of the transform can be adjusted and could be made programmable.

Full Custom Implementation

The present design has been fabricated using standard cells in 1.5μ technology, this has resulted in a physically large ASIC implementation. The use of standard cells also means that unused circuitry is incorporated into the design and the placement and routeing of components may not be ideal. With the further work above into the improved functionality of the adder and sigmoid transform conducted, the design could be optimised to be implemented using a full custom design system, for example CADENCE which is now available in the School of Engineering.

Application of $\mathbf{A_{R-P}}$ to Time Series

Information in the stochastic pulse rate encoded domain is inherently held in a stochastic time series format. The study of the behaviour of A_{R-P} reinforcement learning algorithms when applied to both deterministic and stochastic time series could be investigated. Such work leads to ANN which have recurrence, feedback, incorporated into their structure. This recurrence may be local ie. from a neuron's output back to its own input, or global, ie. from a neuron's output back to the input of neurons preceeding it in the network.

Integration of $\mathbf{A_{R-P}}$.

With an improved set of neuron devices an enlarged network could be considered for construction to enable the integration and evaluation of A_{R-P} reinforcement learning strategies within a complete hardware system. Alternatively, if an accurate software model of the neuron performance including the modelling of the sigmoid is formed a software system could be developed and studied.

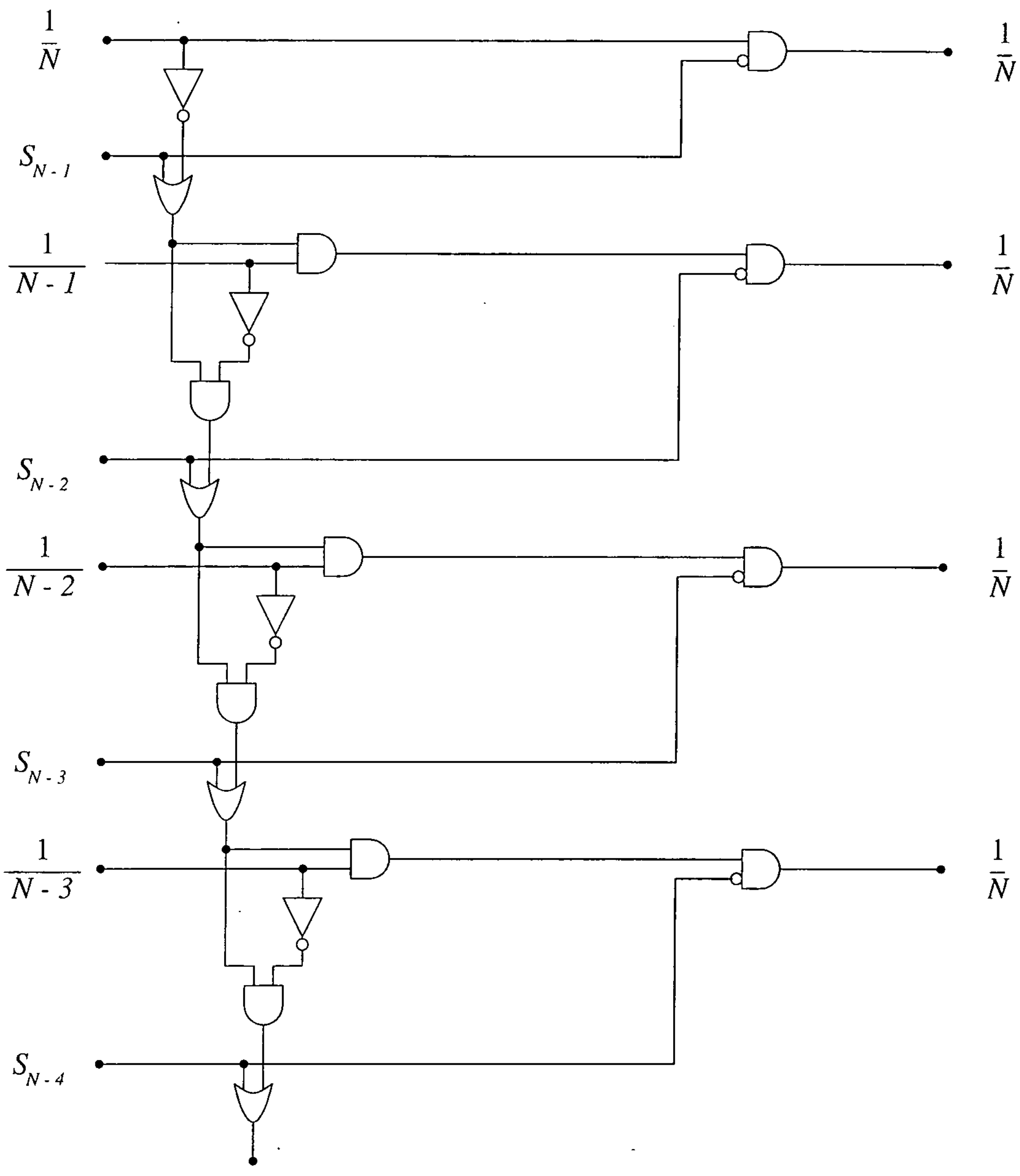


Figure 7.1: A new circuit for the generation of N pulse streams of value $\frac{1}{N}$

Appendix A

Random Number Generation

A.1 Hardware Random Number Generators

Hardware random number generators can be divided into two types, those which implement an algorithm which could be achieved in software and those which are based upon a true random physical process. Use of a physical process can have its drawbacks. The random number generator will often require specific hardware to be used and it is not possible to repeat a sequence unless a record of random numbers generated is maintained. This can cause problems of repeatability when conducting simulation experiments. Physical noise sources are often the basis for random number generators. The following noise sources could be used for the generation of random numbers, thermal sources, noise diodes, gas discharge tubes and radioactive sources.

Johnson, [105], showed that a resistance with no external applied voltage has a measurable noise across its terminals. Nyquist's noise theorem, [106], quantifies this noise for a resistance in a narrow band, Δf, as a function of temperature

$$\bar{u}^2 = \frac{4hf}{(e^{\frac{hf}{kT}-1})} R\Delta f$$

$\bar{u}^2$ is the mean square voltage, h is Planck's constant (6.6×10^{-34} J s^{-1}), k is Boltzmann's constant (1.38×10^{-23} J °K^{-1}), T is absolute temperature, f is frequency and R is the value of resistance. A hot resistance may thus be used to generate a noise signal from which random numbers may be formed. The noise is known as **thermal** noise or **Johnson** noise. A thermal noise source is a primary or absolute source.

Secondary or transfer noise sources examples are diodes or gas discharge tubes. ERNIE, Electronic Random Number Indicator Equipment, [107], the premium bond number generating machine is a practical demonstration that these noise sources can be successfully used for random number generation. Noise waveforms from gas discharge tubes were converted to pulse trains, a random number being formed by counting the number of pulses produced in a given time. ERNIE has been upgraded since its original construction by replacing the gas discharge tubes with diodes.

Diodes, BJTs and FETs can utilise shot noise, thermal noise and avalanche noise to generate adequate levels of noise. A full discussion of the physical processes involved is given by Buckingham, [108]. The formation of diodes or transistors within integrated circuits for generating noise to convert to random pulse sequences is an attractive proposition. Alspector *et al*, [94], states that unfortunately early work using these ideas very high levels of gain were required to use the noise source in transistors. This could lead to cross coupling in the amplifiers, particularly if many sources are integrated onto a single chip. The area required for such a noise source was also considered to be too expensive.

Radioactive decay is a random process. A sequence of random numbers has been generated using a gamma ray source, [109]. The least significant digit of the gamma ray count in a given time period was used as the random number. The distribution of the generated numbers was satisfactory, but implementing a gamma ray source, detector and conversion circuitry poses a problem in a general process. For this reason the experiment was used to create a list of random numbers stored on magnetic tape. A different random number sequence can be gained by starting at a different location on the tape. Tests conducted using these random numbers are repeatable since the tape can be rerun from a given starting point.

Digital shift registers can be used to form PRBS Generators. These will be mentioned only briefly since they are the subject of a more detailed discussion in the main body of the thesis, §5.3. PRBSs are pseudo generators since the output is not strictly random but an output of bits from a linear feedback shift register, LFSR, which have been subjected to modulo two arithmetic. The binary digits output can be used to form random numbers. The sequence in the shift register cycles round, with appropriate feedback selection this will be maximal length ie. the shift register will hold all possible combinations of 1's and 0's, except all 0's, before repeating the sequence. The maximal length of sequence is $2^N - 1$ for a shift register of length N and can be seen to grow exponentially with register length, Figure A.1

A.2 Software Random Number Generators

The fundamental problem with using software to create random numbers is that an algorithm must be used. Numbers formed are therefore deterministic since they are calculated using a precise technique, the numbers only appear to be random. The sequence of numbers will cycle around, by forming generators carefully the period of the cycle can be extremely large so that to all intents and purposes the numbers appear random. Several suitable algorithms have been formed as are detailed below. Algorithms are known as

Pseudo Random Number Generators.

A.2.1 Middle Square Generator

An early algorithm which has been considered is the *middle-square* technique proposed by John von Neumann [110]. The method is to take the square of the previous number and extract the middle digits to form a new random number. If the seed is chosen carefully it is possible to achieve a reasonable sequence. There are several drawbacks, if a zero is generated in the number it tends to be self perpetuating over several numbers, sequences often decay into a cycle of repeating numbers and the seed must be carefully selected. This middle-square technique is considered to be a poor source of random numbers. For a four digit seed of 5781 the first few random numbers generated are as

$$\begin{aligned} 5781 &\rightarrow 33\mathit{4199}961 \\ 4199 &\rightarrow 17\mathit{6316}01 \\ 6316 &\rightarrow 39\mathit{8918}56 \\ 8918 &\rightarrow 79\mathit{5307}24 \\ 5307 &\rightarrow \quad \cdots \end{aligned}$$

A.2.2 Linear Congruential Generators

Most software random number generators are based upon Linear Congruential Generators, LCG's, although this is not the only system possible. An LCG is based upon the following equation,

$$X_{i+1} = (aX_i + c) \bmod m$$

where a is the multiple, c is the increment, and m is the modulus. All three constants are positive integers.

Each generated number is based upon the preceding value. The sequence will cycle around and repeat itself. The length of the sequence depends upon the selection of a, c and m. A table of suitable choices for these values is provided in Numerical Recipes, [111], together with a description of implementations using LCGs. The degree of algorithm complexity and memory usage varies with the quality of result required. The initial value of X is known as the seed and may be set explicitly. It is therefore possible to repeat a sequence of random numbers by re-initialising the seed to the same starting value. Knuth, [110], gives a full description of this technique and the criteria for the selection of parameter values. If c is set to 0 the pseudo-random number generator is called a Multiplicative Linear Congruential Generator, MLCG. L'Ecuyer, [112], describes how several MLCG's may be combined to produce generators with good statistical properties and the ability for the resultant generator to be split into several independent generators.

A.2.3 Lagged-Fibonacci Generators

A sequence based upon a Fibonacci sequence has been suggested. The values are calculated as follows,

$$X_{i+1} = (X_i + X_{i-1}) \bmod m$$

m is the modulus. The sequence period is usually longer than m. The algorithm has been found not to produce sufficiently random results. Extending the above principle such that,

$$X_{i+1} = (X_i + X_{i-k}) \bmod m \tag{A.1}$$

improves the quality of the random numbers generated. Providing k and m are suitably chosen eq.(A.1) can produce adequate random numbers. A large table of past values may need to be maintained for X_i to X_{i-k} which will not lend itself to easy seeding of the sequence as a table of seeds must be formed.

A.2.4 Add–With–Carry and Subtract–With–Borrow

Add–with–carry and subtract–with–borrow random number generator are a relatively recent development introduced by Marsaglia and Zaman, [113]. These generators are related to lagged-Fibonacci generators described above. Properties of these generators include being fast at generating sequences since no multiplications are involved and that the sequences are very long indeed, lengths greater than 2^{500} have been quoted.

The add–with–carry, AWC, sequence is generated as follows,

$$X_i = (X_{i-s} + X_{i-r} + c_i) \bmod b$$

$$c_{i+1} = I(X_{i-s} + X_{i-r} + c_i \geq b)$$

$r > s$ are positive integers called lags, c_i is the carry and I, the indicator function, is 1 or 0 depending upon whether or not the inequality is true or false.

Similarly the subtract–with–borrow, SWB, sequence is generated as follows,

$$X_i = (X_{i-s} - X_{i-r} - c_i) \bmod b$$

$$c_{i+1} = I(X_{i-s} - X_{i-r} - c_i < 0)$$

Again $r > s$ as before, note this time c_i is a borrow.

Early analysis of these types of generator has been promising [113, 114].

This brief list of a few techniques is by no means complete. Active research is taking place into the analysis and generation of random numbers. For the present the LCG and its variants dominate most software implementations due to their easy formulation and understanding.

A.3 Random Number Generator Tests

Given that a technique is being used to form random numbers the quality of the distribution may wish to be ascertained. To achieve this several empirical and statistical tests may be applied to the sequence of numbers. In general these tests are not pass or fail, rather an indication is obtained that a sequence may be more or less random than another.

For a truly random number generator producing an even distribution of numbers, a sequence of 100 zeros in succession is as equally likely as another defined sequence of 100 numbers all of which are different. Yet, if 100 zeros are observed the natural reaction would be to say that the generator was biased when this is not the case at all. Many of the tests conducted upon a sequence of numbers assess the distribution of values within them to determine the quality of randomness. A true random number generator may *fail* such a test for the above stated reasons.

The tests which can be applied to random number sequences may be divided into two main categories statistical and empirical. Some of the main tests in each group are outlined below. A thorough assessment is given by Knuth, [110].

Statistical tests are as follows,

Chi-Square, χ^2, Test. A measure of how improbable an outcome is made. An outcome can be achieved quite naturally but a factor relating its likelihood is calculated. Thus, if an outcome is improbable the test should be repeated to ensure that there is no bias in the generator.

Kolmogorev-Smirnov Test. An assessment on the distribution of outcomes versus the theoretical probabilities of such outcomes are made. This test is particularly useful for distributions where the result can be over a very large or infinite range of values.

Empirical tests are as follows,

Equidistribution Test. This test requires that the numbers are uniformly distributed across an entire range. It is basically a form of Kolmogorev-Smirnov test.

Serial Test. Pairs of successive numbers should be uniformly distributed in an independent manner within the sequence. This test can be extended to triples etc.

Gap Test. The length of gap between occurrences of values is assessed.

Poker or Partition Test. This test classically considers groups of five successive integers as outlined in Table A.1. A χ^2 test is performed on the number of quintuples in each category to determine a performance indication for the generator.

Sequence	Example
All different	abcde
One pair	aabcd
Two pair	aabbc
Three of a kind	aaabc
Full house	aaabb
Four of a kind	aaaab
Five of a kind	aaaaa

Table A.1: Poker or Partition Test Sequence Combinations

Permutation Test. The input sequence is divided into p groups of t elements. Elements in each group can have $t!$ possible relative orderings. The occurrence of each ordering is counted and a χ^2 test applied.

Run Test. A sequence is analysed for *run ups* and *run downs*, ie. an inspection of the lengths of monotonic increasing and decreasing subsequences is made.

This above list of tests is by no means exhaustive. Other tests include analysis of the serial correlation, assessment of the maxima and minima output. A sequence may produce acceptable results with one test but not another. As each test is satisfied the chance that a random number generator produces a good random number sequence is improved.

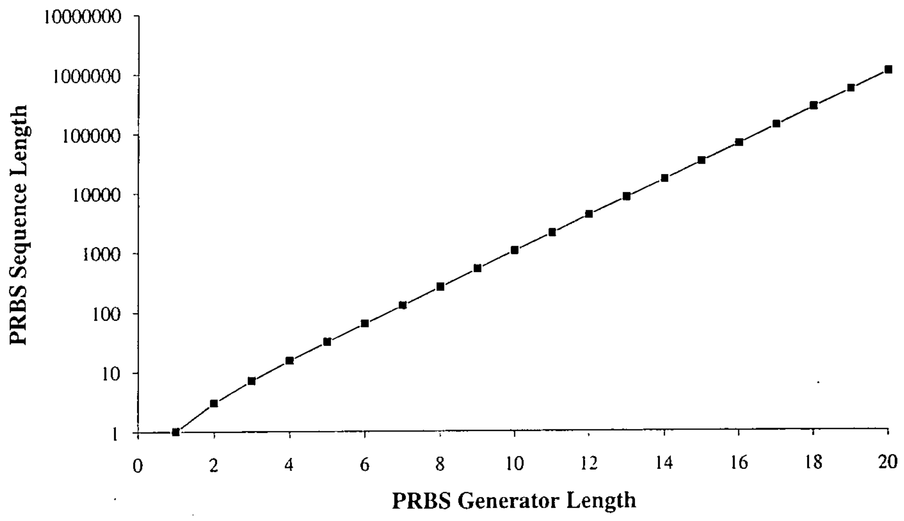


Figure A.1: Maximal binary sequence length vs Shift register length. *The sequence length grows exponentially with an increase in generator length.*

Appendix B

Testing the Quality of the Random Numbers from a PRBS

This appendix describes a series of tests applied to a model of the random number generator constructed and utilised within the artificial neuron chip. The few tests implemented are briefly described in Appendix A and are fully discussed by Knuth [110].

The random number generator selected for implementation was a Pseudo Random Binary Sequence generator of 27-bits. For a 27-bit PRBS feedback is taken from bits 22, 25, 26 and 27 which are **XOR**ed to form the input back into the register to obtain a maximal length sequence. The maximal length sequence of a 27-bit PRBS generator is 134,217,727 bits.

The basic model code for a software PRBS generator is provided in the Appendix C. It consists of a C++ class for a PRBS generator which allows a PRBS to be instantiated and run.

B.1 Correlation Tests

These tests can be performed on two levels and in two forms. The correlation test can be executed either upon the individual bits output from the PRBS or upon the 12-bit random numbers which will be formed from the bit stream in the hardware device. The two forms in which the correlation test can be executed are the auto-correlation and the cross-correlation.

For the 27-bit PRBS generator that is to be used in the artificial neuron chip a simulation was performed to create sequences of bits and sequences of 12-bit random numbers as would occur in the actual hardware. It was ensured that sequences of bits and of 12-bit random numbers were non-overlapping. The auto-correlation and the cross-correlation tests for pulse streams of 1000 bits and a random number sequence of 1000 values have been performed. The results of these four sets of test are shown in Figure B.1 to Figure B.4.

It can be seen from these tests that the degree of correlation in both the auto-corrletation and cross-correlation tests are low except, obviously, for a shift of zero in the auto-correlation test. This is a good indication that the quality of the PRBS bit stream and random numbers is suitable.

B.2 χ^2 Test/Frequency Test

The χ^2 test is a measure of how probable an actual outcome is based upon the expected theoretical outcome. For a uniformly distributed random number the actual distribution of numbers from a run of the generator is compared with the theoretical, ideal, distribution. If the random number can have k values a sequence of n independent random numbers is formed.

Let p_s be the probability each random number is of value s, and let Y_s be the number of such numbers that do actually fall into the category s. A performance measure V is given by eq.(B.1)

$$V = \sum_{1 \leq s \leq k} \frac{(Y_s - np_s)^2}{np_s} \tag{B.1}$$

The numerator can be expanded

$$(Y_s - np_s)^2 = Y_s^2 - 2Y_snp_s + n^2p_s^2$$

and knowing that

$$Y_1 + Y_2 + \cdots + Y_k = n$$

$$p_i + p_2 + \cdots + p_k = 1$$

the following can be derived, eq.(B.2)

$$V = \frac{1}{n} \sum_{1 \leq s \leq k} \left(\frac{Y_s^2}{p_s} \right) - n \tag{B.2}$$

Having calculated the performance measure V it is necessary to determine whether or not such a figure is acceptable. A table of χ^2 distribution values is referred to for ν degrees of freedom where $\nu = k - 1$ as shown in Table B.1. The value in the table, x, is such that V will be less than or equal to x with probability p given that sufficient numbers have been observed. Thus for $\nu = 10$ degrees of freedom the 90% entry of 15.9872 means that V will be greater than this only 10% of the time. In assessing the values of V a figure between 25% and 75% is sort, since for too high values of V doubt is cast upon the likelihood of such an action and for too low values of V the result it too good to be trusted.

The number of categories k may often be large, as is the case for the 12-bit random

ν	$p = 1$	$p = 5$	$p = 10$	$p = 25$	$p = 50$	$p = 75$	$p = 90$	$p = 95$	$p = 99$
1	0.0002	0.0039	0.0158	0.1015	0.4549	1.3233	2.7055	3.8415	6.6349
2	0.0201	0.1026	0.2107	0.5754	1.3863	2.7726	4.6052	5.9915	9.2104
3	0.1148	0.3518	0.5844	1.2125	2.3660	4.1083	6.2514	7.8147	11.3449
4	0.2971	0.7107	1.0636	1.9226	3.3567	5.3853	7.7794	9.4877	13.2767
5	0.5543	1.1455	1.6103	2.6746	4.3515	6.6257	9.2363	11.0705	15.0863
6	0.8721	1.6354	2.2041	3.4546	5.3481	7.8408	10.6446	12.5916	16.8119
7	1.2390	2.1673	2.8331	4.2549	6.3458	9.0371	12.0170	14.0671	18.4753
8	1.6465	2.7326	3.4895	5.0706	7.3441	10.2189	13.3616	15.5073	20.0902
9	2.0879	3.3251	4.1682	5.8988	8.3428	11.3887	14.6837	16.9190	21.6660
10	2.5582	3.9403	4.8652	6.7372	9.3418	12.5489	15.9872	18.3070	23.2093
15	5.2294	7.2609	8.5468	11.0365	14.3389	18.2451	22.3071	24.9958	30.5780
20	8.2604	10.8508	12.4426	15.4518	19.3374	23.8277	28.4120	31.4104	37.5663
25	11.5240	14.6114	16.4734	19.9393	24.3366	29.3388	34.3816	37.6525	44.3140
30	14.9535	18.4927	20.5992	24.4776	29.3360	34.7997	40.2560	43.7730	50.8922
35	18.5089	22.4650	24.7966	29.0540	34.3356	40.2228	46.0588	49.8018	57.3420
40	22.1642	26.5093	29.0505	33.6603	39.3353	45.6160	51.8050	55.7585	63.6908
45	25.9012	30.6123	33.3504	38.2910	44.3351	50.9849	57.5053	61.6562	69.9569
50	29.7067	34.7642	37.6886	42.9421	49.3349	56.3336	63.1671	67.5048	76.1538
75	49.4751	56.0541	59.7946	66.4167	74.3344	82.8581	91.0615	96.2167	106.3929
100	70.0650	77.9294	82.3581	90.1332	99.3341	109.1412	118.4980	124.3421	135.8069

Table B.1: χ^2 distribution values.

numbers, in which instance they may be grouped together into ranges eg. 16 ranges of 256 numbers, 0–255, 256–511, $\cdots$. The distribution within these ranges should be uniform and so the χ^2 test can be performed upon the number of values in each range with probability $p = \frac{1}{r}$ for a number occurring in 1 of the r ranges. This test is known as a Frequency test.

A problem which can occur with the χ^2/Frequency test is selecting the length of the random number sequence to assess. A rule of thumb is that the sequence must be sufficiently long to enable an expected value np_s to be greater than 5. However, if n is too large it may cause local non-random behaviour to be obscured. If n is not large enough a bias which may exist in the random numbers may not be revealed. These tests should be run with varying values of n.

For the 27-bit PRBS generator that is to be used in the artificial neuron chip a simulation was performed to create sequences of 12-bit random numbers as would occur in the actual hardware. The bit pattern set initially in the PRBS generator was derived from the system clock of the host computer, this should ensure a different sub-sequence of the total sequence is assessed each time the test is run. A number of tests were performed for different values of n, 1000 to 1000000, and different degrees of freedom, 10 and 50. For each combination of run length and degrees of freedom 10 runs were evaluated and averaged. The results can be seen in Table B.2 and Table B.3 the mean of which are plotted in Figure B.5.

By reference to Table B.1 it can be seen that the value of V usually lies well within the 25%–75% limits and so with respect to this test upon the 12-bit random numbers they appear to be adequate.

Run Length	1000	10000	100000	1000000	10000000
	7.622	8.806	6.407	12.055	30.681
	8.744	10.193	30.596	13.478	23.425
	8.744	12.234	24.404	11.957	25.432
	6.584	15.002	5.553	24.545	13.249
	7.037	24.041	10.977	8.527	18.204
	13.276	24.067	14.083	8.114	24.860
	13.012	3.919	3.522	13.185	22.865
	5.576	6.727	16.114	9.474	33.344
	5.576	7.143	3.747	11.283	12.823
	8.876	7.603	10.832	12.434	35.689
Mean	8.505	11.973	12.623	12.505	24.057

Table B.2: χ^2 results for distribution of random numbers generated froam a PRBS, 10 degrees of freedom.

RunLength	1000	10000	100000	1000000	10000000
	50.702	44.419	49.110	100.966	378.463
	51.110	62.922	67.799	92.601	389.794
	44.684	53.722	54.500	85.086	333.268
	37.748	35.392	40.311	87.784	380.769
	56.312	58.781	51.866	70.708	363.466
	56.312	44.521	46.852	73.941	365.739
	41.318	54.823	41.002	62.579	382.544
	57.842	37.993	38.548	78.527	412.450
	59.066	60.076	71.452	86.473	406.685
	76.508	52.039	53.699	88.964	360.200
Mean	53.160	50.469	51.514	82.763	377.338

Table B.3: χ^2 results for distribution of random numbers generated froam a PRBS, 50 degrees of freedom

B.3 Gap Test

For the implementation of the Gap test upon a sequence of random numbers the length of gap between a number U within a range and the next occurrence of a number in that range is assessed. A total of n gaps are counted and a χ^2 test performed upon the distribution of these gaps. For a normalised uniform random number generator, given that the number U falls within the range of two numbers α and β, $\alpha \leq U \leq \beta$ and $0 \leq \alpha \leq \beta \leq 1$ the probability that the next number also falls in the same range a gap of zero is

$$p_0 = p = \beta - \alpha$$

For a gap of one

$$p_1 = p(1-p)$$

In fact the probability is a geometric random variable distribution such that for a gap length g

$$p_g = p(1-p)^g$$

Finally for a gap length equivalent to or greater than a user defined maximum length m

$$p_m = (1-p)^m$$

The number of degrees of freedom ν which are applicable for the χ^2 test is the value m the maximum gap length since there are $m+1$ different gap categories.

For the 27-bit PRBS generator that is to be used in the artificial neuron a simulation was performed to create sequences of 12-bit random numbers as per the previous χ^2 test. A number of gap tests were performed for different numbers of gaps from 1000 to 100000 and with two different degrees of freedom 10 and 20. The results can be seen in Table B.4 and Table B.5 with the mean of the 10 separate runs plotted in Figure B.6. By reference to Table B.1 it can be seen that the values of V for the χ^2 distribution test of the gaps usually lies within the 25%–75% limits and so with respect to this test the 12-bit uniform random number generator appears to produce adequately distributed numbers.

B.4 Summary

This appendix has detailed tests applied to a PRBS generator model of 27-bits. The software generates the same bit pattern sequence a hardware realisation of the device. Four basic tests have been conducted upon the random numbers generated using a PRBS generator, auto-correlation, cross-correlation, a χ^2 test and a Gap test for the distribution of values. The basic tests have confirmed that the numbers generated by the use of the

Run Length	1000	2000	5000	10000	20000	50000	100000
	9.252	5.044	4.842	11.918	13.498	9.365	15.441
	11.117	7.101	8.941	4.606	2.626	7.546	12.168
	9.505	21.369	10.669	9.006	18.951	8.007	11.892
	11.013	4.651	9.713	13.748	5.217	10.863	19.351
	11.394	9.334	11.419	10.742	8.208	9.218	4.703
	6.546	11.631	12.571	9.669	15.883	19.226	9.105
	8.067	12.225	9.383	9.950	6.967	4.597	9.276
	4.505	14.166	6.290	6.505	10.014	14.936	7.976
	9.883	14.1658	12.295	11.632	3.672	4.457	8.687
	19.562	12.788	9.400	11.632	16.677	4.179	14.283
Mean	10.084	11.247	9.552	9.941	10.171	9.239	11.288

Table B.4: 10 degrees of freedom for Gap test

Run Length	1000	2000	5000	10000	20000	50000	100000
	4.930	7.814	11.192	16.379	13.949	31.965	51.181
	11.274	14.968	14.597	11.146	11.910	34.132	42.771
	10.656	4.134	6.772	30.478	15.076	38.294	42.532
	6.382	10.265	6.023	7.178	14.432	38.593	63.688
	8.416	9.800	4.493	9.048	13.862	47.532	67.826
	9.528	10.075	2.099	10.206	14.407	21.421	24.303
	5.720	10.015	8.213	19.559	12.621	14.002	41.494
	10.901	9.152	12.544	10.155	16.149	47.094	58.300
	16.304	3.508	10.062	8.182	7.044	11.713	25.393
	13.344	16.937	4.944	12.948	23.906	38.522	54.162
Mean	9.745	9.667	8.094	13.528	14.336	32.327	47.165

Table B.5: 20 degrees of freedom for Gap test

PRBS generator are probably suitably random. The above tests are not exhaustive neither are they conclusive but they do demonstrate that the numbers should be suitably random.

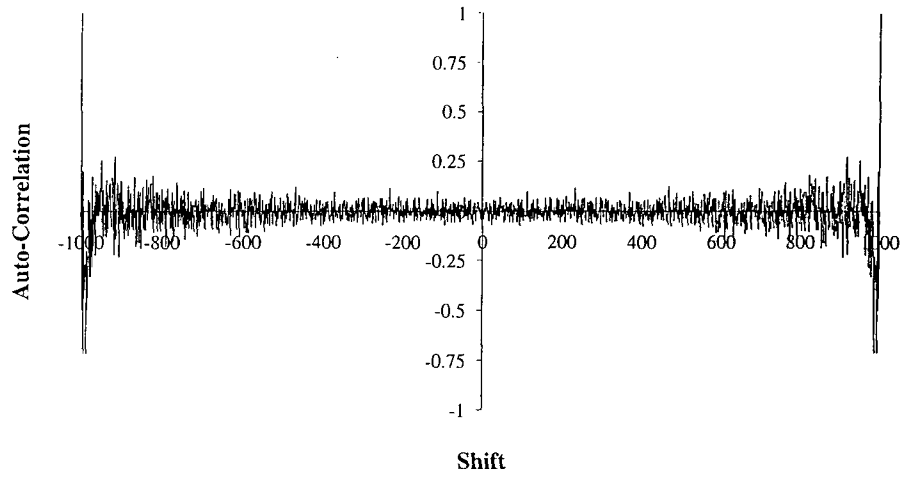


Figure B.1: Auto-correlation for a 1000 bits. *Except for the case of $shift = 0$ the random bit stream exhibits limited correlation.*

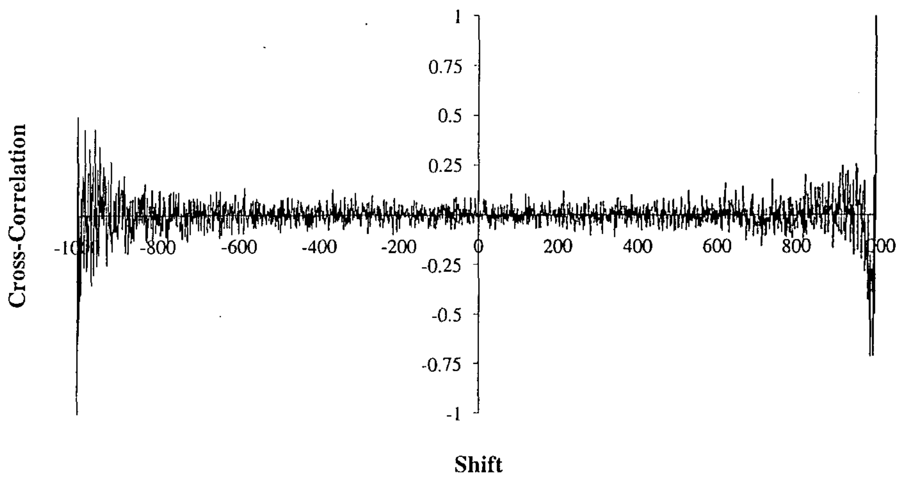


Figure B.2: Cross-correlation for a 1000 bits. *The corrleation between two non-overlapping bit streams is seen to be low*

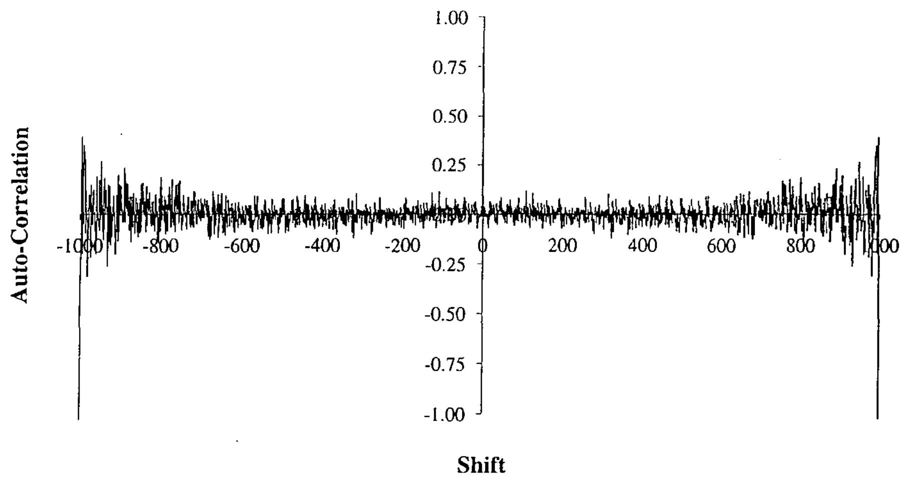


Figure B.3: Auto-correlation for a 1000 numbers. *As per Figure B.1 except for the case of* $shift = 0$ *the 12-bit random number stream exhibits limited correlation.*

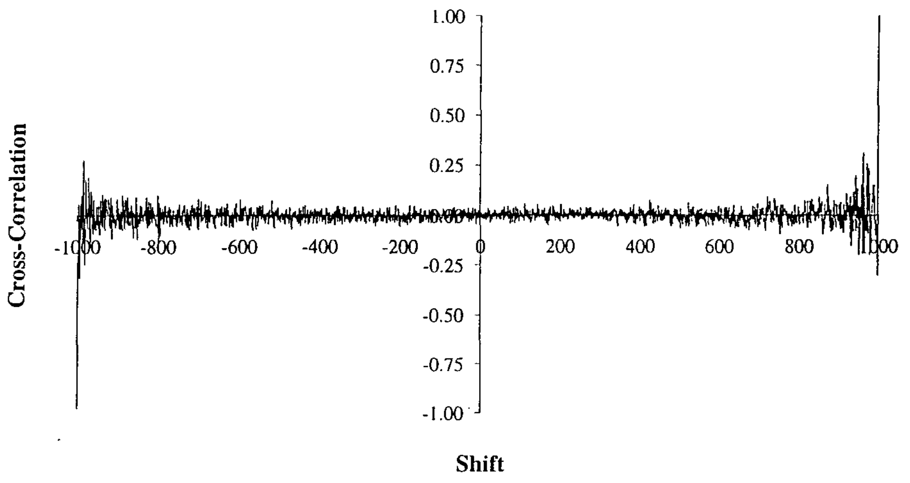


Figure B.4: Cross-correlation for a 1000 numbers. *The cross-correlation between two non-overlapping streams of 12-bit random numbers is low.*

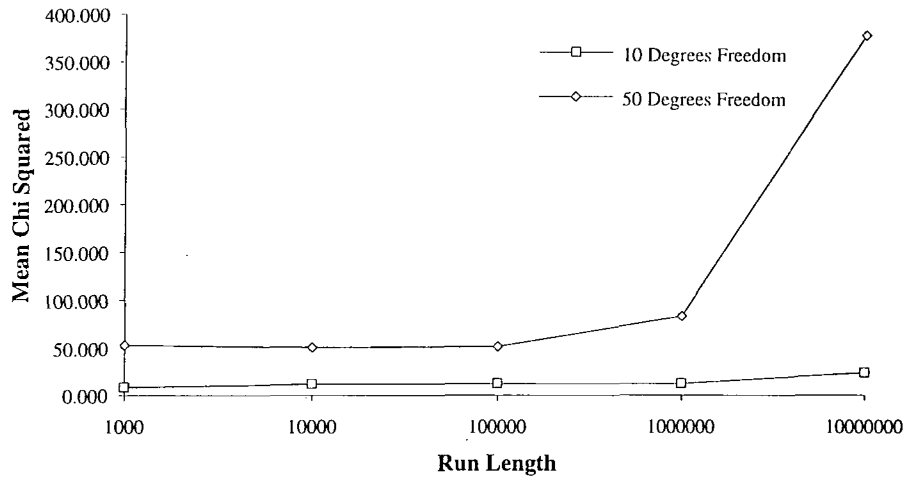


Figure B.5: Mean values of χ^2 test for distribution of random numbers from PRBS generator: 10 and 50 degrees of freedom

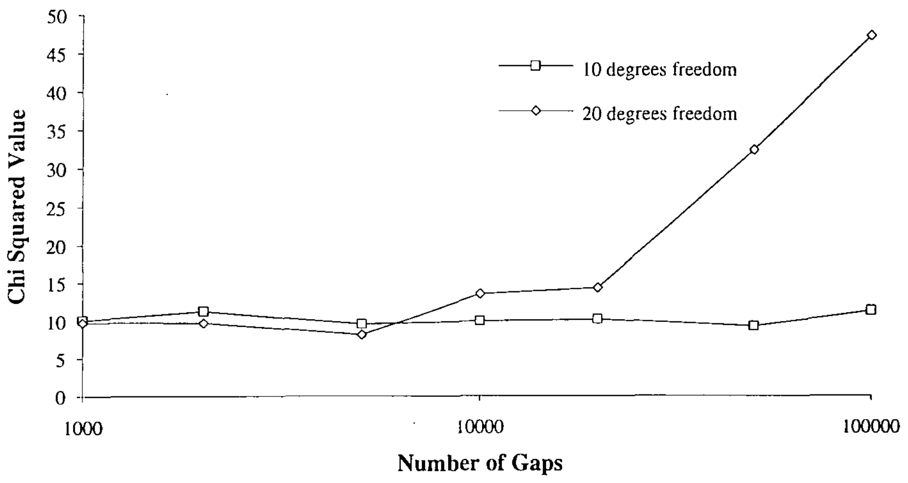


Figure B.6: Mean values of Gap test values for distribution of random numbers from PRBS generator: 10 and 20 degrees of freedom

Appendix C

A C++ PRBS Class

This Class Prbs provides a simple Pseudo Random Binary Sequence (PRBS) generator formed from a Linear Feedback Shift Register (LFSR). The maximum length of the register can be up to 32 bits. A PRBS is used to generate a 0 or a 1 with equal probability. This is achieved by performing modulo 2 arithmetic upon the bit values of the LFSR on which the PRBS is based. The algorithm implementation used to move through the PRBS is that explained in Numerical Recipes, [111], Method 1 is used in this instance. The sequential bits from this type of generator should not be used to form a large random integer or the mantissa of a random float. Knuth, [111], explains that they are not suitable for this purpose. Uses to which these bits can be put are

1. Multiplying a signal randomly by 0 or 1, ±1.
2. A Monte Carlo search of a binary tree where the decision on which direction to branch is the output of the PRBS.

Prbs::Prbs

Function Constructor for Prbs.

Syntax

```
#include "prbs.h"
Prbs RandomBinarySequence(void);
Prbs RandomBinarySequence(const unsigned length,
                          const unsigned long seed);
```

Prototype in prbs.h

Remarks Overloaded constructor for **Prbs** objects. It is usual to specify the *length* and *seed* when first created the object, although they can be set later. The empty constructor is primarily to enable arrays of **Prbs**'s to be created.

Return value none

Prbs::setLength

Function Set the length of the PRBS.

Syntax

```
#include "prbs.h"
int RandomBinarySequence.setLength(const unsigned length);
```

Prototype in prbs.h

Remarks Explicitly set the length of **Prbs** to *length.* If *length* is greater than the maximum size allowed the **Prbs** is set to the maximum possible size and an error returned. It is still possible to use the **Prbs** if required.

Return value If successful returns 0, or returns -1 on an error.

Prbs::seed

Function Seed the PRBS.

Syntax

```
#include "prbs.h"
int RandomBinarySequence.seed(const unsigned long seed);
```

Prototype in prbs.h

Remarks Explicitly set the seed of the **Prbs** to *seed.* If *seed* is greater than the maximum value the this length of PRBS can hold then the seed is set to 1 and an error returned.

Return value If successful returns 0, or returns -1 on an error.

Prbs::advance

Function Advance the generator through its sequence.

Syntax `#include "prbs.h"`

`int RandomBinarySequence.advance(void);`

`int RandomBinarySequence.advance(const unsigned long` *shift*`);`

Prototype in prbs.h

Remarks An overloaded function to advance the **Prbs** through its sequence. This can be a single step or a *shift* number of steps.

Return value Either 0 or 1 is returned, the resultant generator output.

```
// #########
// file:         prbs.h
//
// description: Header file for Class Prbs
//
// author:        John S Glover
// address:       School of Engineering and Computer Science
//                University of Durham
//                South Road
//                Durham
//                DH1 3LE
//                UK
// phone:         +44 91 374 2565
// e-mail:        j.s.glover@durham.ac.uk
//
// #########

#ifndef PRBS_H
#define PRBS_H

#ifndef ULong
typedef unsigned long ULong;
typedef unsigned short UShort;
#endif

const ULong LFSR1 = 1;
const ULong LFSR2 = 2;
const ULong LFSR3 = 4;
const ULong LFSR4 = 8;
const ULong LFSR5 = 16;
const ULong LFSR6 = 32;
const ULong LFSR7 = 64;
const ULong LFSR8 = 128;
const ULong LFSR9 = 256;
const ULong LFSR10 = 512;
const ULong LFSR11 = 1024;
const ULong LFSR12 = 2048;
const ULong LFSR13 = 4096;
const ULong LFSR14 = 8192;
const ULong LFSR15 = 16384;
const ULong LFSR16 = 32768;
const ULong LFSR17 = 65536;
const ULong LFSR18 = 131072;
const ULong LFSR19 = 262144;
const ULong LFSR20 = 5244288;
const ULong LFSR21 = 1048676;
const ULong LFSR22 = 2097152;
const ULong LFSR23 = 4194304;
const ULong LFSR24 = 8388608;
const ULong LFSR25 = 16777216;
const ULong LFSR26 = 33554432;
const ULong LFSR27 = 67108864;
const ULong LFSR28 = 134217728;
const ULong LFSR29 = 268435456;
const ULong LFSR30 = 536870912;
const ULong LFSR31 = 1073741824;
const ULong LFSR32 = 2147483648;
```

```
class Prbs{
  unsigned length;
  ULong prbsValue;
  ULong maxPrbsValue;
  ULong prbsTaps[33];
  ULong randomBit;
  ULong mask;
  const unsigned MaxGeneratorLength = 32;

  public:

    Prbs(void);
    Prbs(const unsigned lengthGenerator, const ULong generatorSeed);

    int Prbs::setLength(const unsigned lengthGenerator);
    int Prbs::seed(const ULong generatorSeed);

    int Prbs::advance(void);
    int Prbs::advance(const ULong shift);
};

#endif

// ##########
// end of file prbs.h
// ##########
// ##########
// file:          prbs.cc
//
// description: function file for Class Prbs generator
//
// author:        John S Glover
// address:       School of Engineering and Computer Science
//                 University of Durham
//                 South Road
//                 Durham
//                 DH1 3LE
//                 UK
// phone:         +44 91 374 2565
// e-mail:        j.s.glover@durham.ac.uk
//
// ##########

static char rcsid[] = "$Id:  prbs.cc%v 1.1 1992/11/01 16:59:38 des3jsg Exp $";

#include <iostream.h>
#include "prbs.h"

// ##########
// member function: Prbs::Prbs()
//
// Constructor for object, all private variables set to 0.
// Useful if creating an array of Prbs's
//
// jsg 28/10/92
// ##########
Prbs::Prbs(void)
{
  length = 0;
  prbsValue = 0;
  maxPrbsValue = 0;
  randomBit = 0;
  mask = 0;
}

// ##########
// member function: Prbs::Prbs()
//
// Constructor for object.
// This is the one to be used in general, requires the size of the PRBS
// to be created and its seed to operate correctly.
//
// jsg 28/10/92
// ##########
Prbs::Prbs(const unsigned lengthGenerator, const ULong generatorSeed)
{
  prbsTaps[0] = 0;
  prbsTaps[1] = LFSR1;
  prbsTaps[2] = LFSR2 + LFSR1;
  prbsTaps[3] = LFSR3 + LFSR2;
  prbsTaps[4] = LFSR4 + LFSR3;
  prbsTaps[5] = LFSR5 + LFSR3;
  prbsTaps[6] = LFSR6 + LFSR5;
  prbsTaps[7] = LFSR7 + LFSR6;
  prbsTaps[8] = LFSR8 + LFSR4 + LFSR5 + LFSR6;
  prbsTaps[9] = LFSR9 + LFSR5;
```

```
    prbsTaps[10] = LFSR10 + LFSR7;
    prbsTaps[11] = LFSR11 + LFSR9;
    prbsTaps[12] = LFSR12 + LFSR6 + LFSR8 + LFSR11;
    prbsTaps[13] = LFSR13 + LFSR9 + LFSR10 + LFSR12;
    prbsTaps[14] = LFSR14 + LFSR9 + LFSR12 + LFSR13;
    prbsTaps[15] = LFSR15 + LFSR14;
    prbsTaps[16] = LFSR16 + LFSR11 + LFSR13 + LFSR14;
    prbsTaps[17] = LFSR17 + LFSR14;
    prbsTaps[18] = LFSR18 + LFSR13 + LFSR16 + LFSR17;
    prbsTaps[19] = LFSR19 + LFSR12 + LFSR17 + LFSR18;
    prbsTaps[20] = LFSR20 + LFSR17;
    prbsTaps[21] = LFSR21 + LFSR19;
    prbsTaps[22] = LFSR22 + LFSR21;
    prbsTaps[23] = LFSR23 + LFSR18;
    prbsTaps[24] = LFSR24 + LFSR20 + LFSR21 + LFSR23;
    prbsTaps[25] = LFSR25 + LFSR3;
    prbsTaps[26] = LFSR26 + LFSR20 + LFSR24 + LFSR25;
    prbsTaps[27] = LFSR27 + LFSR22 + LFSR25 + LFSR26;
    prbsTaps[28] = LFSR28 + LFSR25;
    prbsTaps[29] = LFSR29 + LFSR27;
    prbsTaps[30] = LFSR30 + LFSR24 + LFSR26 + LFSR29;
    prbsTaps[31] = LFSR31 + LFSR28;
    prbsTaps[32] = LFSR32 + LFSR25 + LFSR27 + LFSR29 + LFSR30;

    if (setLength(lengthGenerator) < 0)
      cerr << "error Prbs::Prbs:  incomplete construction" << endl;
    if (seed(generatorSeed) < 0)
      cerr << "error Prbs::Prbs():  incomplete construction" << endl;
}

// #########
// member function: Prbs::setLength()
//
// Set the length of the PRBS to 'lengthGenerator',
// return 0 on success, -1 on failure ie. 'lengthGenerator' too big in which
// case set to 'maxGeneratorLength'.
// Also set 'maxPrbsValue'.
//
// jsg 1/11/92
// #########
int
Prbs::setLength(const unsigned lengthGenerator)
{
  int setLengthReturn = 0;
  maxPrbsValue = 1;

  if (lengthGenerator > MaxGeneratorLength) {
    setLengthReturn = -1;
    cerr << "error Prbs::setLength():  length value out of range" << endl;
    length = MaxGeneratorLength;
    for (unsigned lengthCount = 0; lengthCount < length; lengthCount++) {
      maxPrbsValue <<= 1;
      maxPrbsValue++;
    } // for 'lengthCount'
  }
  else {
    setLengthReturn = 0;
    length = lengthGenerator;
    for (unsigned lengthCount = 0; lengthCount < length; lengthCount++) {
      maxPrbsValue <<= 1;
```

```
        maxPrbsValue++;
      }  // for 'lengthCount'
    }  // if (lengthGenerator > MaxGeneratorLength)

    return setLengthReturn;
}  // function prbs::setLength()

// ##########
// member function: Prbs::seed()
//
// Seed the PRBS with the value of generatorSeed.
// Returns 0 on success, -1 on failure.
// If the seed is too large for the size of PRBS a seed of 1 is used.
//
// jsg 31/10/92
// ##########
int
Prbs::seed(const ULong generatorSeed)
{
    int seedReturn = 0;
    if (generatorSeed > maxPrbsValue) {
      seedReturn = -1;
      cerr << "error Prbs::seed():  seed value out of range" << endl;
      prbsValue = 1;
    }
    else {
      seedReturn = 0;
      prbsValue = generatorSeed;
    }  // if (generatorSeed > maxPrbsValue)

    return seedReturn;
}  // function Prbs::seed()
```

```
// ##########
// member function: Prbs::advance()
//
// Clock PRBS by 1 to advance it once through the sequence.
//
// jsg 29/10/92
// ##########
int
Prbs::advance(void)
{
  randomBit = 0;
  mask = prbsValue & prbsTaps[length];

  for (unsigned bit = 0; bit < length; bit++) {
    mask >>= 1;
    randomBit ∨= mask & 1;
  }  // for 'bit'
  prbsValue = (prbsValue << 1) | randomBit;

  return int(randomBit);
}

// ##########
// member function: Prbs::advance()
//
// Clock PRBS 'shift' times to advance 'shift' steps through the sequence.
//
// jsg 29/10/92
// ##########
int
Prbs::advance(const ULong shift)
{
  for (ULong shiftCount = 0; shiftCount < shift; shiftCount++) {
    randomBit = 0;
    mask = prbsValue & prbsTaps[length];

    for (unsigned bit = 0; bit < length; bit++) {
      mask >>= 1;
      randomBit ∨= mask & 1;
    }  // for 'bit'
    prbsValue = (prbsValue << 1) | randomBit;
  }  // for 'shiftCount'

  return int(randomBit);
}

// ##########
// end of file prbs.cc
// ##########
```

Appendix D

Neuron Test Board Configuration

The connections utilised on the test board for verifying the operation of the fabricated neurons are shown in Table D.1 and Table D.2. The connections relate to those made from the FPC-024 Digital IO cards to the neuron socket and neuron itself. The pins of the fabricated neuron have been illustrated in the main body of this thesis, Figure 6.45. The only additional circuitry required is the provision of a 5V supply suitable to power the artificial neuron device.

In the actual fabricated test board LEDs were connected via buffers to the output lines as a visual feedback of their status, this is not necessary for the operation of the device, conducting the testing of the artificial neuron or as was eventually performed the simulation of a network of six such devices.

Name	Connection	Pin	Name	Connection	Pin
Gnd	CN1-1		Gnd	CN2-1	
Gnd	CN1-2		Gnd	CN2-2	
PA3	CN1-3	27	NC	CN2-3	
NC	CN1-4		NC	CN2-4	
PA2	CN1-5	54	NC	CN2-5	
PA1	CN1-6	58	NC	CN2-6	
PA0	CN1-7	57	NC	CN2-7	
CLK0	CN1-8		NC	CN2-8	
OUT0	CN1-9		NC	CN2-9	
GATE0	CN1-10		NC	CN2-10	
CLK2	CN1-11		NC	CN2-11	
OUT2	CN1-12		NC	CN2-12	
GATE2	CN1-13		PA1	CN2-13	
CLK1	CN1-14		PA0	CN2-14	
GATE1	CN1-15		PA3	CN2-15	
OUT1	CN1-16		PA2	CN2-16	
PA4	CN1-17	28	PA5	CN2-17	
PA5	CN1-18	29	PA4	CN2-18	
PA6	CN1-19	30	PA7	CN2-19	32
PA7	CN1-20	31	PA6	CN2-20	
PC6	CN1-21	73	PC6	CN2-21	50
PC7	CN1-22	74	PC7	CN2-22	51
PC5	CN1-23	72	PC4	CN2-23	48
PC4	CN1-24	71	PC5	CN2-24	49
PC0	CN1-25	67	PC1	CN2-25	45
PC1	CN1-26	68	PC0	CN2-26	44
PC2	CN1-27	69	PB7	CN2-27	43
PB7	CN1-28	66	PC2	CN2-28	46
PC3	CN1-29	70	PB6	CN2-29	42
PB6	CN1-30	65	PC3	CN2-30	47
PB0	CN1-31	59	PB5	CN2-31	39
PB5	CN1-32	64	PB0	CN2-32	34
PB1	CN1-33	60	PB4	CN2-33	38
PB4	CN1-34	63	PB1	CN2-34	35
PB2	CN1-35	61	PB3	CN2-35	37
PB3	CN1-36	62	PB2	CN2-36	36
-5v	CN1-37		-5v	CN2-37	
+5v	CN1-38		+5v	CN2-38	
-12v	CN1-39		-12v	CN2-39	
+12v	CN1-40		+12v	CN2-40	

Table D.1: ASIC connections to FPC-024 digital I/O card 1

Name	Connection	Pin	Name	Connection	Pin
Gnd	CN1-1		Gnd	CN2-1	
Gnd	CN1-2		Gnd	CN2-2	
PA3	CN1-3	78	NC	CN2-3	
NC	CN1-4		NC	CN2-4	
PA2	CN1-5	77	NC	CN2-5	
PA1	CN1-6	76	NC	CN2-6	
PA0	CN1-7	75	NC	CN2-7	
CLK0	CN1-8		NC	CN2-8	
OUT0	CN1-9		NC	CN2-9	
GATE0	CN1-10		NC	CN2-10	
CLK2	CN1-11		NC	CN2-11	
OUT2	CN1-12		NC	CN2-12	
GATE2	CN1-13		PA1	CN2-13	23
CLK1	CN1-14		PA0	CN2-14	22
GATE1	CN1-15		PA3	CN2-15	
OUT1	CN1-16		PA2	CN2-16	
PA4	CN1-17	79	PA5	CN2-17	
PA5	CN1-18	80	PA4	CN2-18	
PA6	CN1-19	81	PA7	CN2-19	2
PA7	CN1-20	82	PA6	CN2-20	
PC6	CN1-21		PC6	CN2-21	20
PC7	CN1-22		PC7	CN2-22	21
PC5	CN1-23		PC4	CN2-23	18
PC4	CN1-24		PC5	CN2-24	19
PC0	CN1-25		PC1	CN2-25	13
PC1	CN1-26		PC0	CN2-26	12
PC2	CN1-27		PB7	CN2-27	10
PB7	CN1-28		PC2	CN2-28	14
PC3	CN1-29		PB6	CN2-29	9
PB6	CN1-30		PC3	CN2-30	17
PB0	CN1-31	83	PB5	CN2-31	8
PB5	CN1-32		PB0	CN2-32	3
PB1	CN1-33	24	PB4	CN2-33	7
PB4	CN1-34		PB1	CN2-34	4
PB2	CN1-35	25	PB3	CN2-35	6
PB3	CN1-36	26	PB2	CN2-36	5
-5v	CN1-37		-5v	CN2-37	
+5v	CN1-38		+5v	CN2-38	
-12v	CN1-39		-12v	CN2-39	
+12v	CN1-40		+12v	CN2-40	

Table D.2: ASIC connections to FPC-024 digital I/O card 2

Appendix E

4–2–4 Encoder/Decoder Board Configuration

The basic schematics for a network of six neurons are illustrated in Figure E.1 and Figure E.2 with the connections made between this system and the two FPC-024 digital IO boards detailed in Table E.2 and Table E.3.

Each neuron, Neuron X, is configured similarly but there are variations between hidden layer and output neurons. The two hidden layer neurons, Neurons 1 and 2, have four input lines which are shared and five independent up/down lines each for driving the weight register counters. The four output layer neurons, Neurons 3, 4, 5 and 6, have two shared input lines taken from the outputs of the hidden layer neurons and three independent up/down lines each for driving the weight register counters. For each neuron the remaining input lines are commoned together and driven as one with a value of zero, IN X where X is a given neuron. Similarly for each neuron the remaining up/down lines are commoned together and driven as one with a value of zero, UD X, so that neuron weight does not change.

Name	Description
RST	Reset
CLK	Clock
R/W	Read/Write line for each individual neuron driven by a combination of the global R/W and the selection of the neuron.
ADDR 0–7	Address lines, 0–4 select the appropriate weight register and 5–7 select the appropriate neuron
In 0-3	The four input driving pulse sequences for the network.
UD $X0$–n	Up/down signal lines for neuron X and weight registers 0 to n
Init 0–11	Initialising weight value, if being loaded.
WghtOut	Encoded pulse trains for encoded weights.
SumOut	Result of weighted summation.
Out X	Result of sigmoid transform circuit for neuron x.
IN X	Remaining unused input lines for a neuron X commoned together and driven as one.
UD X	Remaining unused up/down lines for a neuron X commoned together and driven as one.

Table E.1: Description of signal line naming convention.

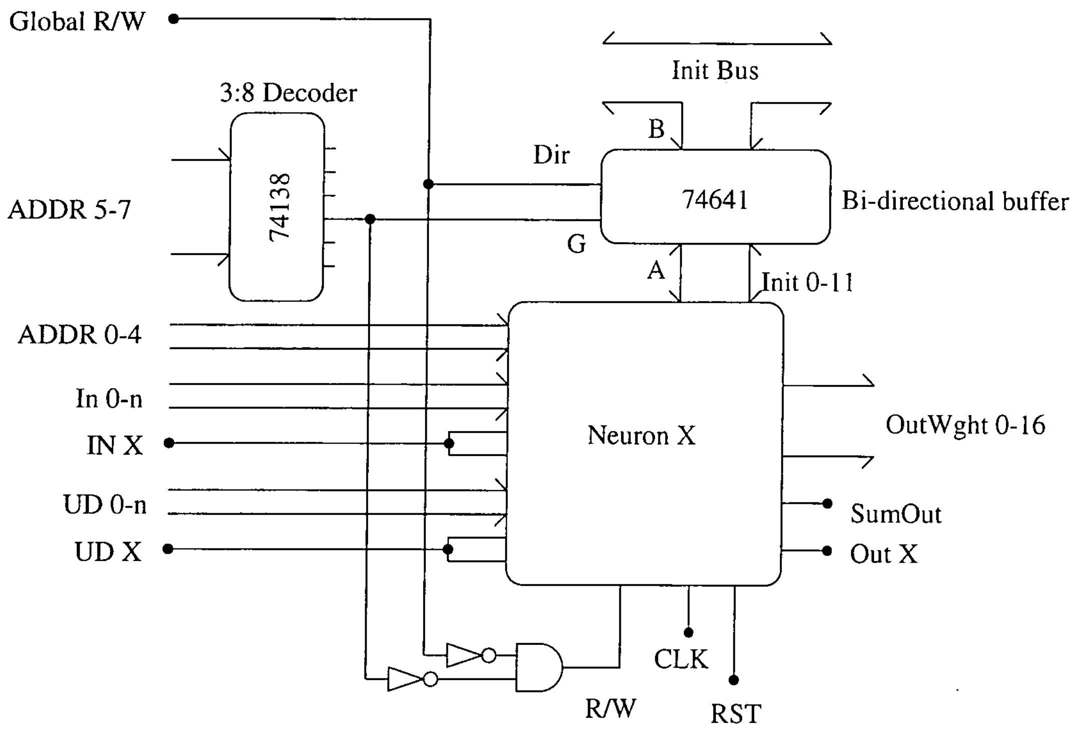


Figure E.1: An individual neuron configuration, Neuron X. *For the 4-2-4 encoder/decoder six circuit are required connected to a bus on a motherboard. Input In 0–n are the input lines either from the outside or from the preceding layer.*

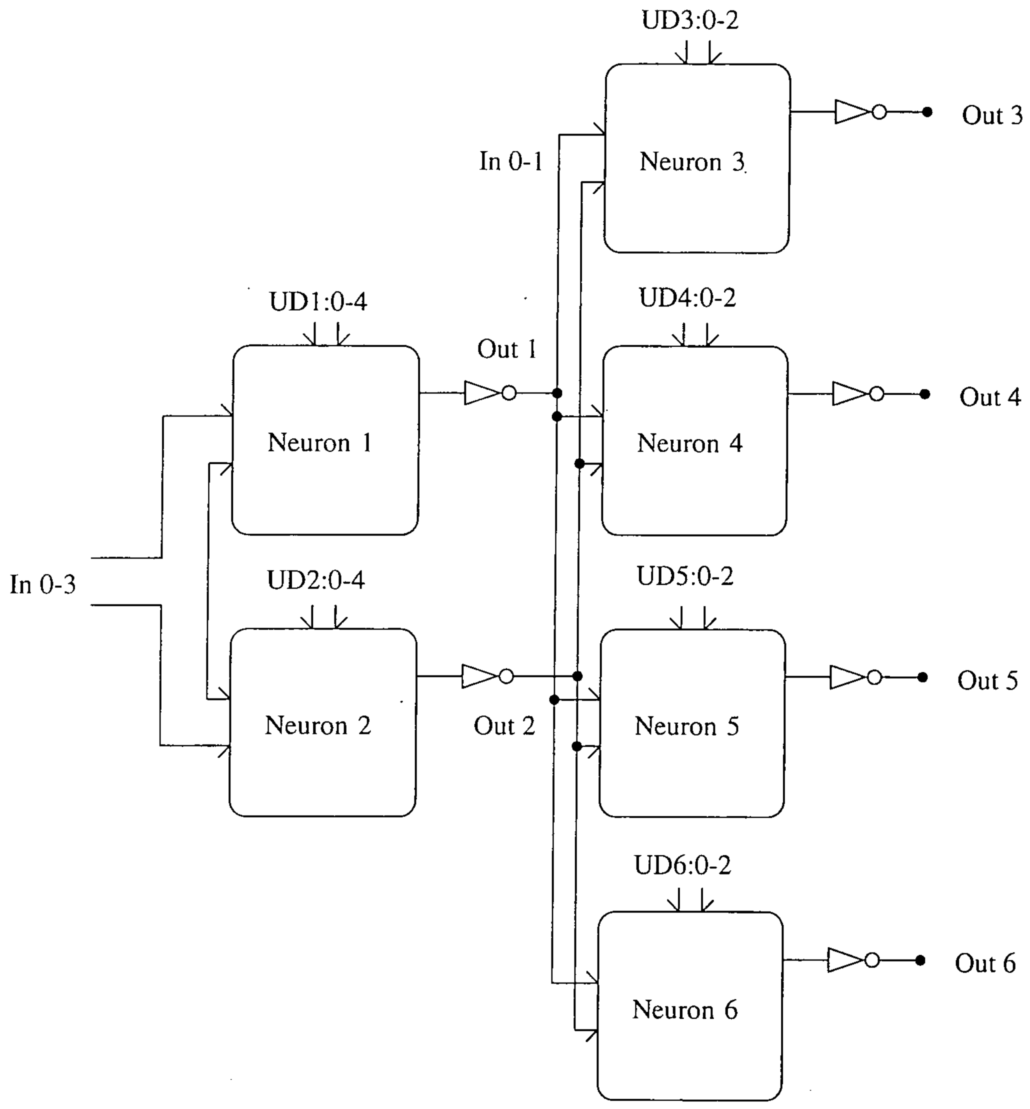


Figure E.2: Encoder/Decoder system configuration of six neurons. *In 0–3 are the input values to the system from the outside world, while In 0–1 are the output from the hidden layer fed into the output layer. All neuron outputs have an inverter on them to correct the value of the sigmoid transform output.*

Name	Connection	Pin	Name	Connection	Pin
Gnd	CN1-1		Gnd	CN2-1	
Gnd	CN1-2		Gnd	CN2-2	
PA3	CN1-3	Addr 3	NC	CN2-3	
NC	CN1-4		NC	CN2-4	
PA2	CN1-5	Addr 2	NC	CN2-5	
PA1	CN1-6	Addr 1	NC	CN2-6	
PA0	CN1-7	Addr 0	NC	CN2-7	
CLK0	CN1-8		NC	CN2-8	
OUT0	CN1-9		NC	CN2-9	
GATE0	CN1-10		NC	CN2-10	
CLK2	CN1-11		NC	CN2-11	
OUT2	CN1-12		NC	CN2-12	
GATE2	CN1-13		PA1	CN2-13	UD 1.1
CLK1	CN1-14		PA0	CN2-14	UD 1.0
GATE1	CN1-15		PA3	CN2-15	UD 1.3
OUT1	CN1-16		PA2	CN2-16	UD 1.2
PA4	CN1-17	Addr 4	PA5	CN2-17	
PA5	CN1-18	Addr 5	PA4	CN2-18	UD 1.4
PA6	CN1-19	Addr 6	PA7	CN2-19	
PA7	CN1-20	Addr 7	PA6	CN2-20	
PC6	CN1-21		PC6	CN2-21	
PC7	CN1-22		PC7	CN2-22	
PC5	CN1-23	Out 6	PC4	CN2-23	SumOut 5
PC4	CN1-24	Out 5	PC5	CN2-24	SumOut 6
PC0	CN1-25	Out 1	PC1	CN2-25	SumOut 2
PC1	CN1-26	Out 2	PC0	CN2-26	SumOut 1
PC2	CN1-27	Out 3	PB7	CN2-27	
PB7	CN1-28	RW	PC2	CN2-28	SumOut 3
PC3	CN1-29	Out 4	PB6	CN2-29	
PB6	CN1-30	RST	PC3	CN2-30	SumOut 4
PB0	CN1-31	In 0	PB5	CN2-31	
PB5	CN1-32	CLK	PB0	CN2-32	UD 2.0
PB1	CN1-33	In 1	PB4	CN2-33	UD 2.4
PB4	CN1-34		PB1	CN2-34	UD 2.1
PB2	CN1-35	In 2	PB3	CN2-35	UD 2.3
PB3	CN1-36	In 3	PB2	CN2-36	UD 2.2
-5v	CN1-37		-5v	CN2-37	
+5v	CN1-38		+5v	CN2-38	
-12v	CN1-39		-12v	CN2-39	
+12v	CN1-40		+12v	CN2-40	

Table E.2: Motherboard connections to digital I/O card 1

Name	Connection	Pin	Name	Connection	Pin
Gnd	CN1-1		Gnd	CN2-1	
Gnd	CN1-2		Gnd	CN2-2	
PA3	CN1-3	Init 3	NC	CN2-3	
NC	CN1-4		NC	CN2-4	
PA2	CN1-5	Init 2	NC	CN2-5	
PA1	CN1-6	Init 1	NC	CN2-6	
PA0	CN1-7	Init 0	NC	CN2-7	
CLK0	CN1-8		NC	CN2-8	
OUT0	CN1-9		NC	CN2-9	
GATE0	CN1-10		NC	CN2-10	
CLK2	CN1-11		NC	CN2-11	
OUT2	CN1-12		NC	CN2-12	
GATE2	CN1-13		PA1	CN2-13	UD 3.1
CLK1	CN1-14		PA0	CN2-14	UD 3.0
GATE1	CN1-15		PA3	CN2-15	
OUT1	CN1-16		PA2	CN2-16	UD 3.2
PA4	CN1-17	Init 4	PA5	CN2-17	UD 4.1
PA5	CN1-18	Init 5	PA4	CN2-18	UD 4.0
PA6	CN1-19	Init 6	PA7	CN2-19	
PA7	CN1-20	Init 7	PA6	CN2-20	UD 4.2
PC6	CN1-21		PC6	CN2-21	IN 6
PC7	CN1-22		PC7	CN2-22	UD 6
PC5	CN1-23		PC4	CN2-23	IN 5
PC4	CN1-24		PC5	CN2-24	UD 5
PC0	CN1-25	IN 1	PC1	CN2-25	UD 3
PC1	CN1-26	UD 1	PC0	CN2-26	IN 3
PC2	CN1-27	IN 2	PB7	CN2-27	
PB7	CN1-28		PC2	CN2-28	IN 4
PC3	CN1-29	UD 2	PB6	CN2-29	UD 6.2
PB6	CN1-30		PC3	CN2-30	UD 4
PB0	CN1-31	Init 8	PB5	CN2-31	UD 6.1
PB5	CN1-32		PB0	CN2-32	UD 5.0
PB1	CN1-33	Init 9	PB4	CN2-33	UD 6.0
PB4	CN1-34		PB1	CN2-34	UD 5.1
PB2	CN1-35	Init 10	PB3	CN2-35	UD 5.2
PB3	CN1-36	Init 11	PB2	CN2-36	
-5v	CN1-37		-5v	CN2-37	
+5v	CN1-38		+5v	CN2-38	
-12v	CN1-39		-12v	CN2-39	
+12v	CN1-40		+12v	CN2-40	

Table E.3: Motherboard connections to digital I/O card 2

Bibliography

[1] S H Tsao. Generation of delayed replicas of maximal-length linear binary sequences. *Procs. of the IEE*, 111(11):1803–1806, Nov 1964.

[2] A G Barto and M I Jordan. Gradient following without back-propagation in layered networks. In *Proc. 1st Annual Conference on Neural Networks*, 1987.

[3] W S McCulloch and W Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[4] F Rosenblatt. The perceptron: A probalistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[5] P K Simpson. *Artificial Neural Systems: foundations, paradigms, applications and implementations.* Pergamon Press, 1990. ISBN 0–08–037894–3.

[6] J A Hertz, A S Krogh, and R G Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991. ISBN 0–201–51560-1.

[7] B Widrow and M E Hoff. Adaptive switching circuits. *1960 IRE Western Electric Show and Convention Record*, (4):96–104, Aug 1960.

[8] M Minsky and S Papert. *Perceptrons: An introduction to computational geometry.* MIT Press Massachusetts, 1969. ISBN 0–262–13043–2.

[9] R P Lippman. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(2):4–22, April 1987.

[10] G Mirchandani and W Cao. On hidden nodes for neural nets. *IEEE Trans. on Circuits and Systems*, 36(5):661–664, May 1989.

[11] S-C Huang and Y-H Huang. Bounds on the number of hidden neurons in multilayer perceptrons. *IEEE Trans. on Neural Networks*, 2(1):47–55, Jan 1991.

[12] J Makhoul, A El-Jaroudi, and R Schwartz. Partioning capabilities of two-layer neural networks. *IEEE Trans. on Signal Processing*, 39(6):1435–1440, June 1991.

[13] P Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[14] D B Parker. Learning logic. Technical Report Rechnical Report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, 1985.

[15] D E Rumelhart, J L McClelland, and the PDP Research Group. *Parallel Distributed Processing*, volume 1, chapter 8. Learning Internal Representation of Error Propagation, pages 318–364. The MIT Press, 1986. ISBN 0–262–18120–7.

[16] T P Vogl, J K Mangis, A K Rigler, W T Zink, and D L Alkon. Accelerating the convergence of the bacl-propagation method. *Biological Cybernetics*, 59:257–263, 1988.

[17] R S Scalero and N Tepedelenlioglu. A fast new algorithm for training feedforward neural networks. *IEEE Trans. on Signal Processing*, 40(1):202–210, January 1992.

[18] G R Little, S C Gustafson, and R A Senn. Generalization of the backpropagation neural network learning algorithm to permit complex weights. *Applied Optics*, 29(11):1591–1592, April 1990.

[19] H Leung and S Haykin. The complex backpropagation algorithm. *IEEE Trans. on Signal Processing*, 39(9):2101–2104, Sept 1991.

[20] N Benvenuto and F Piazza. On the complex backpropogation algorithm. *IEEE Trans. on Signal Processing*, 40(4):967–969, April 1992.

[21] G M Georgiou and C Koutsougeras. Complex domain backpropagation. *IEEE Trans. on Circuits and Systems—II: Analog and Digital Signal Processing*, 39(5):330–334, May 1992.

[22] T Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982.

[23] T Kohonen. Analysis of a simple self-organizing process. *Biological Cybernetics*, 44:135–140, 1982.

[24] T Kohonen. The self-organising map. *Procs. of the IEEE*, 78(9):1464–1480, Sept 1990.

[25] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Procs. National Academy of Sciences of the USA*, 79:2554–2558, April 1982.

[26] J J Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Procs. National Academy of Sciences of the USA*, 81:3088–3092, May 1984.

[27] A Murray and L Tarassenko. *Analogue Neural VLSI: A pulse stream approach.* Chapman & Hall, 1994. ISBN 0–412–45060–7.

[28] J J Hopfield and D W Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[29] D W Tank and J J Hopfield. Simple 'neural' optimization networks: An A/D convertor, signal decision circuit and a linear programming circuit. *IEEE Trans. on Circuits and Systems*, 33:533–541, 1986.

[30] D O Hebb. *The Organization of Behaviour: A Neuropsychological Theory.* John Wiley & Sons, 1949.

[31] D H Ackley, G E Hinton, and T J Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985.

[32] B Widrow, N K Gupta, and S Maitra. Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Trans. on Systems, Man and Cybernetics*, 3(5):455–465, Sept 1973.

[33] A G Barto, R S Sutton, and P S Brouwer. Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40:201–211, 1981.

[34] A G Barto and R S Sutton. Landmark learning: An illustration of associative search. *Biological Cybernetics*, 42:1–8, 1981.

[35] A G Barto, C W Anderson, and R S Sutton. Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics*, 43:175–185, 1982.

[36] A G Barto, R S Sutton, and G W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. on Systems, Man and Cybernetics*, 13(5):834–846, Sept/Oct 1983.

[37] K S Narendra and M A L Thathachar. Learning automata — a survey. *IEEE Trans. on Systems, Man and Cybernetics*, 4(4):323–334, July 1974.

[38] L E Atlas and Y Suziki. Digital systems for artificial neural networks. *IEEE Circuits and Devices Magazine*, 5(6):20–24, Nov 1989.

[39] H P Graf and L D Jackal. Analog electronic neural network circuits. *IEEE Circuits and Devices Magazine*, 5(4):44-49, July 1989.

[40] S Y Foo, L R Anderson, and Y Takefufi. Analog components for the VLSI of neural networks. *IEEE Circuits and Devices Magazine*, 6(4):18–26, July 1990.

[41] C Mead. *Analog VLSI and Neural Systems.* Addison-Wesley, 1989.

[42] M S Tomlinson Jr, D J Walker, and M A Sivilotti. A digital neural network architecture for VLSI. In *Proc. IEEE IJCNN*, volume II, pages 545–550, San Diego, CA, 1990.

[43] R A Leaver. *Stochastic Arrays and Learning Networks.* PhD thesis, University of Durham, 1988.

[44] F G Stremler. *Introduction to Communication Systems.* Addison-Wesley, 2nd edition, 1982. ISBN 0–201–07259–9.

[45] A F Murray and A J W Smith. Asynchronous arithmetic for VLSI neural systems. *Electronics Letters*, 23(12):642, June 1987.

[46] A F Murray and A J W Smith. Asynchronous VLSI neural networks using pulse-stream arithmetic. *IEEE Jour. of Solid-State Circuits*, 23(3):688–697, June 1988.

[47] M Brownlow, L Tarassenko, A F Murray, A Hamilton, I S Han, and H M Reekie. Pulse-firing neural chips for hundreds of neurons. *Advances in Neural Information Processing Systems*, pages 785–792, 1990.

[48] M J Brownlow, L Tarassenko, and A F Murray. Analogue computation using VLSI neural network devices. *Electronics Letters*, 26(16):1297–1299, 1990.

[49] A F Murray, D Delcorso, and L Tarassenko. Pulse-stream VLSI neural networks mixing analog and digital techniques. *IEEE Trans. on Neural Networks*, 2(2):222–229, 1991.

[50] A Hamilton, A F Murray, D J Baxter, S Churcher, H M Reekie, and L Tarassenko. Integrated pulse stream neural networks: Results, issues, and pointers. *IEEE Trans. on Neural Networks*, 3(3):385–393, 1992.

[51] J N Tombs, L Tarassenko, and A F Murray. A novel analogue VLSI design for multilayer networks. *IEE Proceedings—F Radar and Signal Processing*, 139(6):426–430, 1992.

[52] P M Daniell, W A J Waller, and D L Bisset. An implementation of fully analogue sum-of-product neural models in VLSI. In *Proc. 1st IEE Conference on Artificial Neural Networks*, pages 52–56, 1989.

[53] B R Gaines. Stochastic computing systems. In J T Tou, editor, *Advances in Information System Science*, volume 2, pages 37–172. Plenum Press, 1969.

[54] D Nguyen and F Holt. Stochastic processing in a neural network. In *IEEE First International Conference on Neural Networks*, pages 281–291, 1987.

[55] D E Van Den Bout and T K Miller III. A stochastic architecture for neural nets. In *IEEE 2nd International Conference on Neural Networks*, pages 481–488, 1988.

[56] D E Van Den Bout and T K Miller III. A digital architecture employing stochasticism for the simulation of Hopfield neural nets. *IEEE Trans. on Circuits and Systems*, 36(5):732–738, May 1989.

[57] W Banzhaf. On a simple stochastic neuron-like unit. *Biological Cybernetics*, 60:153–160, 1988.

[58] H Eguchi, T Furuta, H Horiguchi, and S Oteki. Neural network hardware with learning function using pulse-density modulation. *Electronics and Communications in Japan*, 74(11):66–74, 1991.

[59] H Eguchi, T Furuta, H Horiguchi, S Oteki, and T Kitaguchi. Neural network LSI chip with on-chip learning. In *Proc. IEEE IJCNN*, volume I, pages 453–456. Seattle, 1991.

[60] Y Kondo and Y Sawada. Functional abilities of a stochastic logic neural network. *IEEE Trans. on Neural Networks*, 3(3):434–443, May 1992.

[61] P Hyland. *On the Implementation of Neural Networks Using Stochastic Architecture.* PhD thesis, University College of North Wales. Bangor, 1992.

[62] *80170NX Electronically Trainable Analog Neural Network*, June 1991. Experimental datasheet.

[63] J Brauch, S M Tam, M A Holler, and A L Shmurun. Analog VLSI neural networks for impact signal processing. *IEEE Micro*, 12(6):34–45, Dec 1992.

[64] *NI1000 Recognition Accelerator*, 1994.

[65] *MT19003, Neural Instruction Set Processor*, May 1994.

[66] A F Murray. Pulse arithmetic in VLSI neural networks. *IEEE Micro*, 9(6):64–74, Dec 1989.

[67] A F Murray. Silicon implementations of neural networks. *IEE Proceedings—F Radar and Signal Processing*, 138(1):3–12, 1991.

[68] J L Meador, A Wu, C Cole, and N Nintunze an P Chintrakulchai. Programmable impulse neural circuits. *IEEE Trans. on Neural Networks*, 2(1):101–109, Jan 1991.

[69] N E Cotter and O N Mian. A pulsed neural network capable of universal approximation. *IEEE Trans. on Neural Networks*, 3(2):308–314, Mar 1992.

[70] J E Tomberg and K K K Kaski. Pulse-density modulation technique in VLSI implementations of neural network algorithms. *IEEE Jour. of Solid-State Circuits*, 25(5):1277–1286, Oct 1990.

[71] P S Churchland and T J Sejnowski. *The Computational Brain.* The MIT Press, 1992. ISBN 0–262–03188–4.

[72] B R Gaines. A stochastic analog computer. Standard Telecommunication Laboratories, Internal Memorandum, 1–10, Dec 1965.

[73] B R Gaines and J H Andreae. A learning machine in the context of the general control problem. In *Proceedings of the 3rd Congress of the International Federation for Automatic Control.* Butterworths, London, 1966.

[74] W J Poppelbaum and C Afusco. Noise computer. Technical report, University of Illinois: Dept. of Computer Science, 1965. Quarterly Technical Progress Reports April 1965 – January 1966.

[75] P Mars and W J Poppelbaum. *Stochastic and Deterministic Averaging Processors.* Peter Peregrinus Ltd., 1981. ISBN 0–906048–44–3.

[76] S M Ross. *Introduction to Probability Models.* Academic Press, 4th edition, 1989. ISBN 0–12–598464–2.

[77] A J Miller. *Digital Stochastic Computation.* PhD thesis, University of Aberdeen, 1976.

[78] Y Taki, H Miyakawa, M Hatori, and S Namba. Even-shift orthogonal sequences. *IEEE Trans. on Information Theory*, 15(2):295–300, Mar 1969.

[79] M J E Golay. Complementary series. *IRE Trans. on Information Theory*, 7(2):82–87, April 1961.

[80] E Kreyszig. *Advanced Engineering Mathematics.* Wiley, 7th edition, 1993. ISBN 0–471–59989-1.

[81] T Izumi. Fast generation of a white and normal random signal. *IEEE Trans. on Instrumentation and Measurement*, 37(2):316–318, June 1988.

[82] A C Davies. Probability distribution of noislike waveforms generated by a digital technique. *Electronics Letters*, 4(19):421–423, 1968.

[83] I Izumi. A method of generating multidimensional normally distributed random signals. *Trans. Japan Soc. Instrum. Control Eng.*, 13:517–522, 1977.

[84] J G Kalbfleisch. *Probability and Statistical Inference: 1.* Springer-Verlog, 1979.

[85] R C Tausworthe. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19:201–209, 1965.

[86] S W Golomb. *Shift Register Sequences.* Aegean Park Press, 2nd edition, 1982. ISBN 0–89412–048–4.

[87] R B Pearson, J L Richardson, and D Toussaint. A fast processor for monte-carlo simulation. *Journal of Computational Physics*, 51(2):241–249, Aug 1983.

[88] A Hoogland, J Spaa, B Selman, and A Compagner. A special-purpose processor for the monte carlo simulation of ising spin systems. *Journal of Computational Physics*, 51(2):250–260, Aug 1983.

[89] J Saarinen, J Tomberg, L Vehmanen, and K Kaski. VLSI implementation of Tausworthe random number generator for parallel processing environment. *IEE Proceedings—E*, 138(3):138–146, May 1991.

[90] A C Davies. Delayed versions of maximal-length linear binary sequences. *Electronics Letters*, 1(3):61–62, 1965.

[91] A C Davies. Further notes on delayed versions of linear binary sequences. *Electronics Letters*, 1(7):190–191, 1965.

[92] A N Van Luyn. Shift-register connections for delayed versions of m-sequences. *Electronics Letters*, 14(22):713–715, 1978.

[93] A B Gardiner. Logic PRBS delay calculator and delayed-version generator with automatic delay-changing facility. *Electronics Letters*, 1(5):123–124, 1965.

[94] J Alspector, J W Gannett, S Haber, M B Parker, and R Chu. A VLSI efficient technique for generating multiple uncorrelated noise sources and its application to stochastic neural networks. *IEEE Trans. on Circuits and Systems*, 38(1):109–123, Jan 1991.

[95] P J M van Laarhoven and E H L Arts. *Simulated Annealing: Theory and Applications.* D Reidel Publishing Company, 1987. ISBN 90–277–2513–6.

[96] N Metropolis, A W Rosenbluth, M N Rosenbluth, and A H Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, June 1953.

[97] S Kirkpatrick, C D Gelatt Jr., and M P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[98] J H Holland. *Adaption in Natural and Artificial Systems.* 1975.

[99] L Ingber and B Rosen. Genetic algorithms and very fast simulated reannealing: A comparison. *Mathematical and Computer Modelling*, 16(11):87–100, 1992.

[100] L Ingber. Simulated annealing: Practice versus theory. *Statistics and Computing*, 1993. To be Published.

[101] European Silicon Structures. *Solo 1400 Reference Manuals and Databooks*, June 1991. Release V3.1.

[102] E J Watson. Primitive polynomials (mod 2). *Mathematics of Computation*, 16:368–369, 1962.

[103] B Holdsworth. *Digital Logic Design.* 2nd edition, 1987. ISBN 0408015667.

[104] C H Roth. *Fundementals of Logic Design.* 3rd edition, 1985. ISBN 0314852921.

[105] J B Johnson. Thermal agitation of electricity in conductors. *Physical Review*, 32:97–109, July 1928.

[106] H Nyquist. Thermal agitation of electric charge in conductors. *Physical Review*, 32:110–113, July 1928.

[107] W E Thomson. ERNIE - a mathematical and statistical analysis. *Jour. of the Royal Statistical Society. Series A*, 122(3):301–333, 1959.

[108] M J Buckingham. *Noise in Electronic Devices and Systems.* Ellis Horwood Ltd., 1983.

[109] H Inoue, H Kumahora, Y Yoshizawa, M Ichimura, and O Miyatake. Random numbers generated by a physical device. *Applied Statistics*, 32(1):115–120, 1983.

[110] D E Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 2nd edition, 1981. ISBN 0-201-03822-6.

[111] W H Press, B P Flannery, S A Teukolsky, and W T Vetterling. *Numerical recipes in C. The art of scientific computing.* Cambridge University Press, 1988. ISBN 0-521-35465-X.

[112] P L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749,774, June 1988.

[113] G Marsaglia and A Zaman. A new class of random number generator. *The Annals of Applied Probability*, 1(3):462–480, Aug 1991.

[114] S Tezuka and P L'Ecuyer. Analysis of add–with–carry and subtract–with–borrow generators. In *Procs. of the 1991 Winter Simulation Conference*, pages 443–447, Dec 1992. ISBN 0-7803-0798-4.